VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BUILDING A PULL – BASED CONTENT DELIVERY NETWORK ON GCP

BCSE408P

CLOUD COMPUTING

G1 + TG1 SLOT

SUBMITTED TO

DR. ANUSOOYA

ARVIN SAMUEL A (23BCE1283)

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to everyone who contributed to the completion of this comprehensive report. First and foremost, I extend my deepest appreciation to my Cloud Computing professor, Dr. Anusooya, whose guidance and support were invaluable throughout the process. Your insightful feedback and encouragement have truly enhanced the quality of this work.

I am also indebted to my classmates and friends who provided valuable insights and perspectives during discussions and brainstorming sessions. Your input greatly enriched the content of this report.

Furthermore, I would like to thank my family for their unwavering support and understanding, especially during times when I needed to dedicate extra hours to working. Your patience and encouragement kept me motivated to strive for excellence.

Thank you to everyone who contributed in any way to the completion of this report. Your support has been instrumental in making this project a success.

TABLE OF CONTENTS

ABSTRACT

In today's hyper-connected digital era, the speed, availability, and reliability of content delivery have become essential for the success of any web-based system. As web users become increasingly global, distributing content to end-users with minimal latency and maximum efficiency poses a critical challenge. Content Delivery Networks (CDNs) are vital infrastructure solutions to this problem, offering geographically distributed caching and optimized content routing. However, traditional CDN services such as Cloudflare or Akamai are proprietary, expensive, and not always transparent or educationally accessible for individual developers or students seeking to learn and experiment with content distribution techniques.

This project aims to bridge that gap by designing and implementing a pull-based, scalable CDN using modern cloud-native tools on Google Cloud Platform (GCP). The primary objective is to understand and apply concepts of distributed caching, containerization, serverless compute, global routing, and event-driven communication using a cohesive, real-world architecture. The project was done entirely as a solo effort, and it leverages key GCP services such as Google Kubernetes Engine (GKE), Cloud Run, Artifact Registry, Redis, Pub/Sub, and Global Load Balancing. The backend is built in Go, chosen for its efficiency, concurrency support, and suitability for scalable web systems.

In this architecture, a single origin server deployed on GKE (in the asia-south-1 region) serves as the authoritative source for all content, while seven globally distributed edge nodes—hosted on Cloud Run—act as CDN nodes. When a client requests content, the nearest node first checks its local Redis cache. If the content is not found, it is fetched from the main server, cached locally, and then returned to the client. Each node also uses hash-based deduplication in Redis to reduce redundant storage and employs Time-To-Live (TTL) mechanisms to expire stale content. Additionally, Redis Pub/Sub enables near-instant cache update propagation between nodes.

This project not only demonstrates a deep understanding of cloud architecture but also addresses practical challenges like private-public service communication, load balancing across regions, and real-time content synchronization. It simulates a production-grade CDN environment using only GCP's native services, making it both cost-efficient and scalable. The implementation also includes a demo, performance testing via Insomnia, and outlines several areas for future expansion, such as dynamic edge node scaling and support for a wider variety of file types.

INTRODUCTION:

The exponential growth of internet usage and digital content consumption has created a high demand for systems that can deliver content to users in an efficient, low-latency, and scalable manner. From videos and images to APIs and software binaries, content is being accessed around the world with increasing expectations for speed and availability. At the heart of enabling this experience lies the Content Delivery Network (CDN)—a system designed to distribute and deliver digital content based on the geographical location of the user, origin of the webpage, and the content delivery servers.

Conventional CDNs typically operate on a push-based model, where content is proactively distributed to multiple edge servers. While this approach has proven effective, it often involves complex setup, vendor lock-in, and opaque mechanisms. These systems are typically managed by large-scale companies and expose limited interfaces to developers, reducing the opportunities for learning and innovation.

This project takes a different approach—a pull-based CDN, which only caches content on-demand. When a client requests a file that is not present in the edge cache, the request is forwarded to the main server, which then supplies the content to the node. This content is stored locally using Redis and served in subsequent requests. This method not only saves bandwidth and reduces unnecessary data replication but also mimics real-world architectures of distributed content platforms, making it an ideal subject for academic exploration in a Cloud Computing course.

The project is implemented on Google Cloud Platform (GCP) to take full advantage of its suite of managed services, particularly GKE for orchestrating containerized applications, Cloud Run for serverless compute at the edge, Redis for in-memory caching, Pub/Sub for event-driven updates, and Global Load Balancing to ensure the best node is always selected for a user's request.

By working on this project, the goal was not just to build a technically functional system but to explore the design trade-offs, deployment challenges, and performance considerations inherent to global-scale cloud applications. Additionally, this project aims to provide an extensible foundation for future improvements such as intelligent caching, predictive content fetching, and broader content-type support. The end result is a robust, scalable, and educational CDN prototype that demonstrates key cloud computing principles in action.

LITERATURE REVIEW:

The concept and design of Content Delivery Networks (CDNs) have been extensively studied in both academia and industry. The underlying objective across these works has been to improve content availability, reduce latency, and increase the scalability of web services. This section discusses three key papers that informed and inspired the design decisions taken in this project.

1. "A Survey on Content Delivery Networks" – Pathan, A.-S.-K., & Buyya, R. (2007)

Source: Communications of the ACM

This foundational survey provides a comprehensive overview of CDN architectures, taxonomy, and challenges. The paper distinguishes between push-based and pull-based CDNs, highlighting that while push-based models reduce latency through pre-distribution, pull-based models are more efficient in terms of storage and bandwidth usage—since content is only cached when requested.

This directly influenced the choice of implementing a pull-based CDN architecture in our project. As highlighted in the survey, pull-based CDNs are more adaptive in environments with high content dynamism and varied user demand, such as personal blogs, APIs, or user-generated content systems. The paper also details the importance of cache placement strategies and cache invalidation—two areas that were implemented using TTL and Pub/Sub mechanisms in our system.

Furthermore, the survey advocates for geographical distribution of cache nodes to minimize round-trip latency—a principle followed in this project by deploying 7 nodes globally using Cloud Run's regional serverless capabilities.

2. "Distributed Hash Tables for Content Distribution" – Stoica, I., et al. (2003)

Source: Proceedings of the ACM SIGCOMM

This paper discusses the use of Distributed Hash Tables (DHTs) and content-addressable storage in large-scale distributed systems. While the project does not use a DHT in the formal sense, the idea of content hashing for deduplication and identification, as used in DHTs, has been central to this project's design.

Inspired by the mechanisms outlined in the paper, the CDN uses a two-level Redis-based cache, where the first level maps a key (such as a filename or content ID) to a SHA-256 hash, and the second level maps the hash to the actual content. This system ensures that duplicate content is never stored multiple times, optimizing memory usage and improving cache lookup speeds.

The paper also explores decentralized content discovery and lookup latency, which motivated the design of node-level independence and the distributed cache model in this implementation.

3. "An Event-Driven Model for Cache Consistency in Edge Networks" – Kangasharju, J., Ross, K. W., & Rejaie, R. (2002)

Source: IEEE INFOCOM

This paper presents a model where event-based communication is used to maintain cache consistency in edge networks. One of the challenges in CDN systems is ensuring that edge caches do not serve stale or outdated content. While pull-based CDNs reduce this issue by design, even they require mechanisms for cache invalidation or synchronization when updates occur at the origin.

The paper's insights into event-driven cache consistency were directly applied using Redis Pub/Sub in this project. When an edge node updates its cache after a pull or content creation event, a message is published to a Pub/Sub topic, notifying other nodes of the change. These nodes can then invalidate or update their local caches accordingly. This reduces the possibility of serving outdated content and maintains a reasonable level of eventual consistency across the network, without tight coupling between nodes.

This approach is highly scalable and decoupled, as recommended by the authors, and fits well within the serverless microservices model employed in the project.

SYSTEM CONFIGURATION:

Development and Deployment Environment:

- **Language**: Go 1.20+
- **Containerization**: Docker
- **Operating System**: Ubuntu 22.04 (for development), Debian containers on GCP
- **Cloud Platform**: Google Cloud Platform (GCP)

Cloud Services Used:

Component	**Service**
Main Server	Google Kubernetes Engine (GKE)
Edge Nodes	Google Cloud Run
Load Balancing	GCP Global Load Balancer
Cache	Redis (deployed per node)
Messaging	Redis Pub/Sub
CI/CD	Artifact Registry + Docker
Testing	Insomnia API Client

SYSTEM ARCHITECTURE:

The system architecture of the pull-based Content Delivery Network (CDN) is designed to showcase the use of distributed cloud-native technologies, efficient caching strategies, and global scalability using Google Cloud Platform (GCP). The core objective of the architecture is to serve user requests from the nearest possible location with minimal latency, using intelligent content fetching, deduplication, and update propagation. The entire architecture is modular, scalable, and built using containerized microservices, leveraging both Kubernetes and serverless compute platforms.

1. High-Level Architectural Components

The architecture is logically divided into the following layers:

- **Main Origin Server (Centralized Backend Layer)**
- **Edge Nodes (Geographically Distributed Frontend Layer)**
- **Caching and Storage Layer (Redis + GCP Storage)**
- **Update Propagation and Event Handling Layer (Redis Pub/Sub)**
- **Global Load Balancer Layer (HTTP/HTTPS Load Balancer + Serverless NEGs)**

2. Main Server (Origin Layer)

- **Deployment Platform:** Google Kubernetes Engine (GKE)
- **Location:** asia-south1 (Mumbai)
- **Purpose:** This acts as the authoritative source of content. When a client requests content not available in the edge cache, the request is routed to this origin server. It is responsible for:
    - Generating content hashes using SHA-256.
    - Serving the content to requesting edge nodes.
    - Storing content in Redis (used by both main and edge nodes).
    - Publishing update events to Pub/Sub for notifying other nodes of new or modified content.

The GKE cluster is deployed as a **private cluster** to isolate it from direct internet access, improving security. Only authenticated edge nodes within the GCP VPC can make requests to the main server using secure internal communication.

3. Edge Nodes (Content Distribution Layer)

- **Deployment Platform:** Google Cloud Run (fully managed serverless containers)
- **Purpose:** These nodes are responsible for serving client requests as close to the user as possible. If the content is present in the local Redis cache, it is served instantly. Otherwise, the node forwards a request to the main server, stores the response in cache, and serves it to the user.
- **Features:**
    - Independent and stateless
    - Subscribe to Pub/Sub for content updates
    - Implements **TTL-based cache expiration**
    - Uses **deduplication-aware caching** (key → hash, hash → value)

- **Deployment Regions of the Edge Nodes:**
    1. us-central1 (Iowa, USA) – For North America
    2. southamerica-east1 (São Paulo, Brazil) – For South America
    3. europe-west2 (London, UK) – For Europe
    4. africa-south1 (Johannesburg, South Africa) – For Africa
    5. asia-south2 (Delhi, India) – For North-Central Asia
    6. asia-northeast1 (Tokyo, Japan) – For East Asia
    7. australia-southeast1 (Sydney, Australia) – For Oceania

This regional deployment ensures **low-latency delivery** by placing nodes closer to users globally, aligning with the goal of **geo-distributed service availability**.

4. Redis-Based Caching and Deduplication Strategy

All nodes, including the origin server, use Redis for caching content. Redis is configured to store content using a **deduplicated two-step mapping**:

- key → hash (e.g., "about.png" → "f23a93...")
- hash → base64-encoded content

This prevents redundant storage of duplicate content across nodes, greatly reducing memory usage, and allowing more content to be cached efficiently. Each node has its own isolated Redis instance to maintain independence.

Additionally, **TTL (Time-to-Live)** values are configured per content entry to allow automatic cache expiry after a certain time, helping balance freshness and memory efficiency.

5. Pub/Sub Based Event Propagation

To enable **eventual consistency** and efficient update propagation across nodes, the project uses **Redis Cloud Pub/Sub**. The main server publishes messages to a topic every time:

- A content is uploaded
- An existing content is updated

Each edge node is subscribed to the topic and listens for these messages. On receiving a relevant message, a node:

- Checks if the cached content is stale
- Invalidates or updates the local cache if needed

This mechanism enables **decoupled and asynchronous** cache invalidation, improving responsiveness while preserving consistency across the system.

6. Global Load Balancer (Ingress Layer)

The CDN is accessible via a **single global IP address** thanks to **Google Cloud HTTP(S) Load Balancer** configured with **Serverless NEGs** (Network Endpoint Groups). This load balancer:

- Routes incoming requests to the nearest healthy edge node based on latency and location
- Offers SSL termination and HTTPS support
- Provides observability and monitoring via GCP

This setup ensures optimal routing for all clients worldwide and abstracts the complexity of dealing with individual node endpoints.

7. Inter-Service Communication

All communication between:

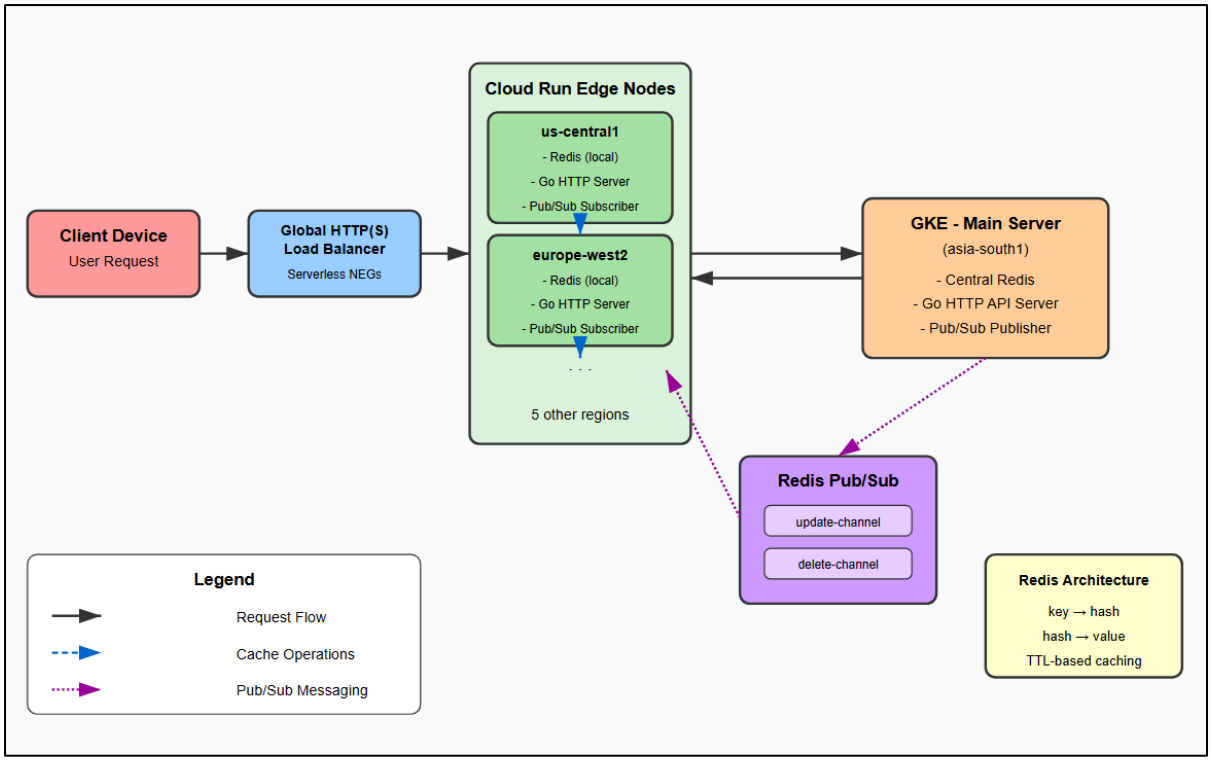
- Cloud Run edge nodes and GKE server
- Nodes and Redis
- Nodes and Pub/Sub

happens through authenticated and encrypted channels. Service accounts with **least privilege access** are configured for security. The internal service discovery and network routing use **VPC connectors** where needed, particularly for enabling private communication between GKE and Cloud Run.

8. Content Flow Summary

The step-by-step overview of a typical request cycle is as follows

1. A client sends a content request to the CDN's global IP.
2. The global load balancer routes the request to the nearest edge node (e.g., europe-west2).
3. The edge node checks its Redis cache:
    - If content is present and valid, it serves immediately.
    - If not, it forwards the request to the GKE origin server.
4. The origin server serves the content, which the node then caches locally.
5. Any new content upload or change triggers a Pub/Sub message from the origin.
6. All other edge nodes listen to the message and act accordingly (invalidate or fetch new).


Cloud Run Edge Nodes
us-central1
- Redis (local)
- Go HTTP Server
- Pub/Sub Subscriber
europe-west2
- Redis (local)
- Go HTTP Server
- Pub/Sub Subscriber
. . .
5 other regions
Client Device
User Request
Global HTTP(S)
Load Balancer
Serverless NEGs
GKE - Main Server
(asia-south1)
- Central Redis
- Go HTTP API Server
- Pub/Sub Publisher
Redis Pub/Sub
update-channel
delete-channel
Legend
Request Flow
Cache Operations
Pub/Sub Messaging
Redis Architecture
key → hash
hash → value
TTL-based caching

IMPLEMENTATION:

The implementation of the pull-based CDN revolves around a **microservices architecture**, where each component is implemented as a containerized Go service. The architecture is composed of two main subsystems: the **Main Server (origin server)** and multiple **Edge Nodes** distributed globally. Both types of services are backed by Redis for content caching and share a common structure to facilitate extensibility and consistency.

The implementation is split across four Go source files:

- server_main.go and server_redis.go (for the Main Server)
- node_main.go and node_redis.go (for the Edge Nodes)

Main Server Implementation

server_main.go

This file contains the HTTP server for the origin backend. It defines handlers for three main operations:

- **GET**: Fetch content using a key
- **POST**: Store new content (or update existing)
- **DELETE**: Remove content

Each handler interacts with Redis through wrapper functions implemented in server_redis.go. These functions handle cache logic and ensure deduplication.

The server supports the following REST endpoints:

- GET /<key> – Returns the decoded content associated with the key
- POST /<key> – Accepts a JSON body with a base64 string and stores it
- DELETE /<key> – Removes the key from Redis and notifies all edge nodes via Pub/Sub

In main(), the server initializes:

- Local Redis connection
- Central Redis (for Pub/Sub communication)
- HTTP listener

It also gracefully handles shutdown signals.

server_redis.go

This file handles Redis logic for the origin server.

- Set(key, value):
  - Hashes the content using **SHA-512**
  - Stores the actual content at hash → value
  - Stores the logical key at key → hash
  - Prevents redundant data storage using hash checks
- Get(key):
  - Looks up the hash from the key
  - Then retrieves the actual content using the hash
- Delete(key):
  - Removes the key
  - Deletes the associated hash **only if no other key references it**
- Publish(channel, message):
  - Sends update/delete messages to the **Pub/Sub** channels:
    - "update-channel" for new/updated content
    - "delete-channel" for content deletion

Edge Node Implementation

node_main.go

This file runs the HTTP server for edge nodes, which accept the same three HTTP methods:

- GET /<key> – First checks local cache; on miss, fetches from main server
- POST /<key> – Stores data locally and pushes to the main server
- DELETE /<key> – Deletes content both locally and from the main server

Additionally, in the main() function, it sets up:

- Redis and Pub/Sub initialization
- Subscription to both update-channel and delete-channel
- A goroutine that continuously listens for messages and updates or deletes local cache accordingly

This mechanism ensures **eventual consistency** and allows the edge node to **stay updated passively** after an origin-side change.

node_redis.go

This file contains the core caching and communication logic for edge nodes.

- NodeGet(key):
  - Attempts to fetch from local Redis
  - If not present, sends a GET request to the main server
  - On success, caches the content locally with a **TTL of 24 hours**

- NodeSet(key, value):
    - Stores the content locally
    - Sends a POST to the main server to ensure synchronization
- NodeDelete(key):
    - Deletes the key from the local cache
    - Sends a DELETE request to the main server

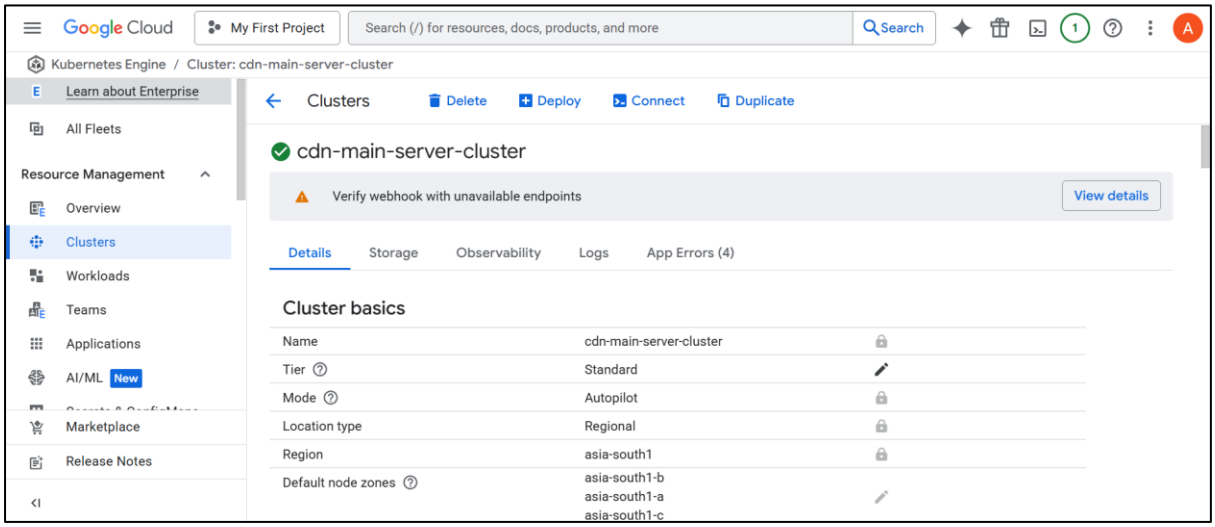
Redis logic is very similar to the server-side:

- Deduplication via key → hash and hash → value
- TTL is applied only on the edge nodes, ensuring **time-bound caching**

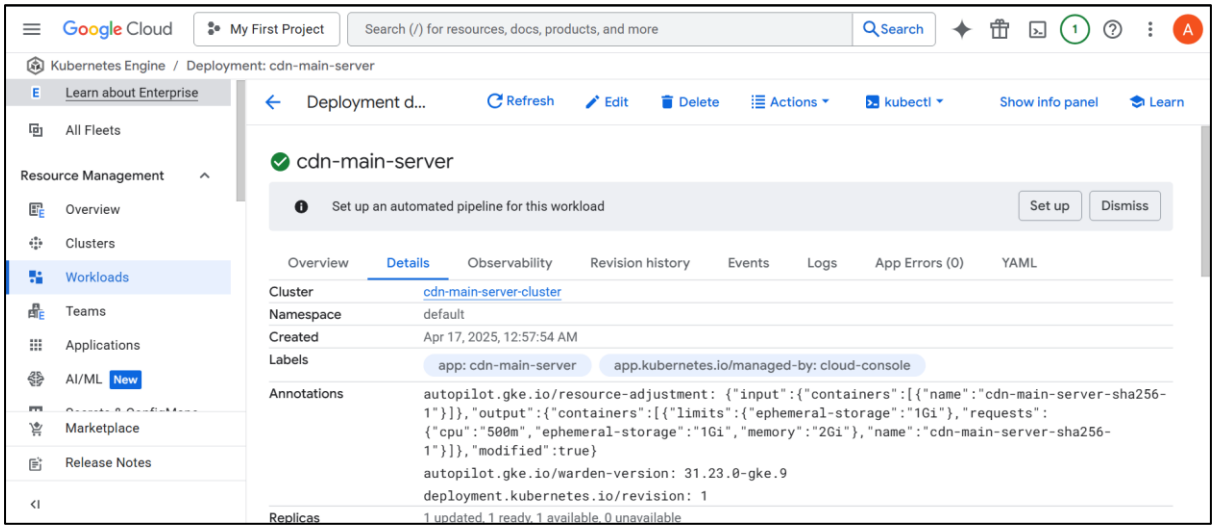
The node also uses the same Pub/Sub logic as the server to:

- Receive update-channel messages and update its cache accordingly
- Receive delete-channel messages and remove keys accordingly

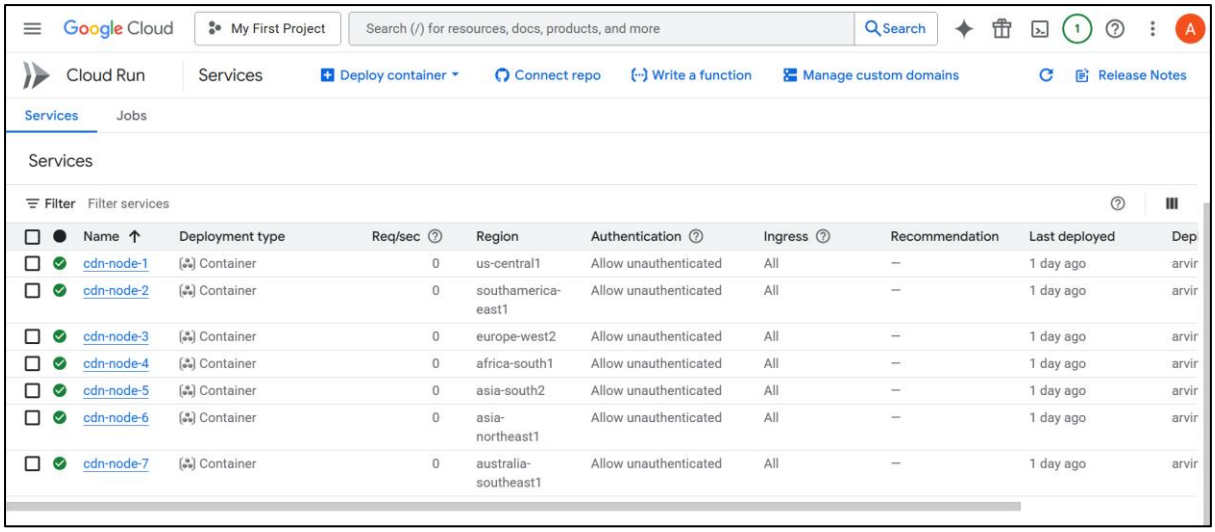
SCREENSHOTS:



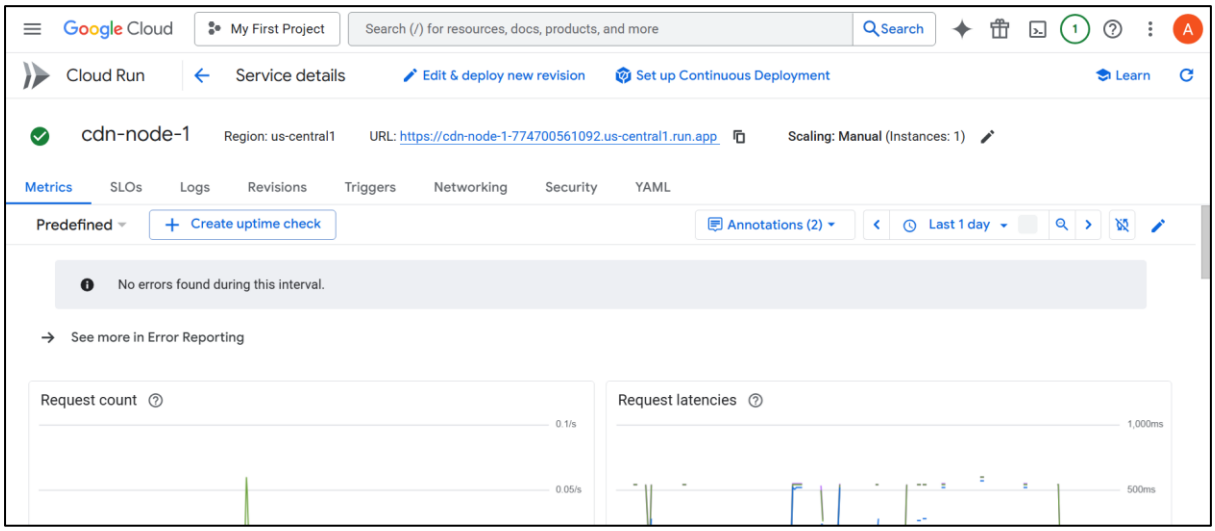
The Dashboard of the GKE Cluster of the Main Server



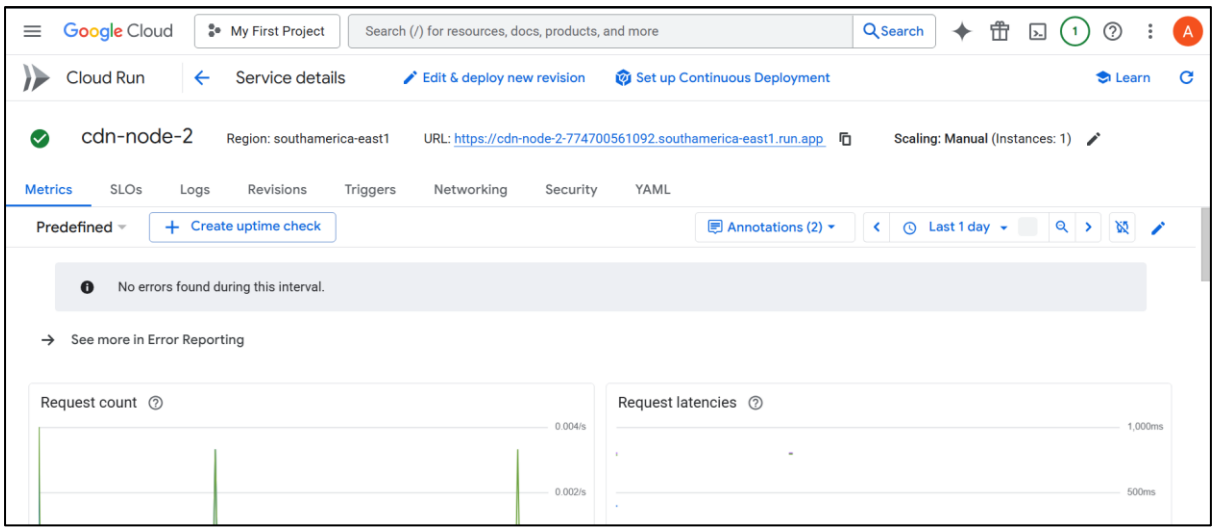
The Dashboard of the Main Server workload inside the Main Service Cluster



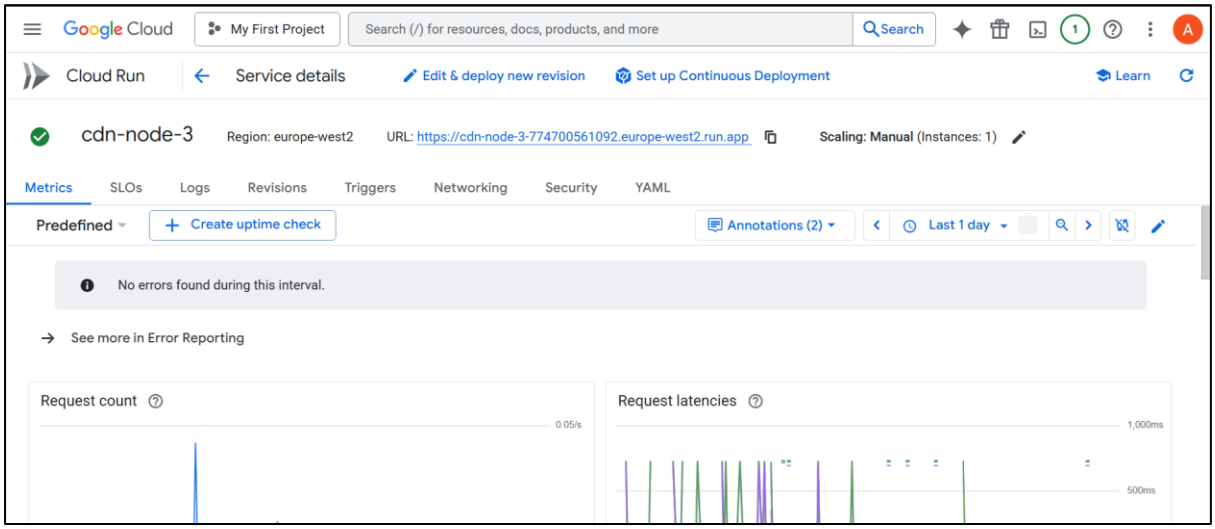
The Cloud Run dashboard showing all the 7 deployed nodes in various regions



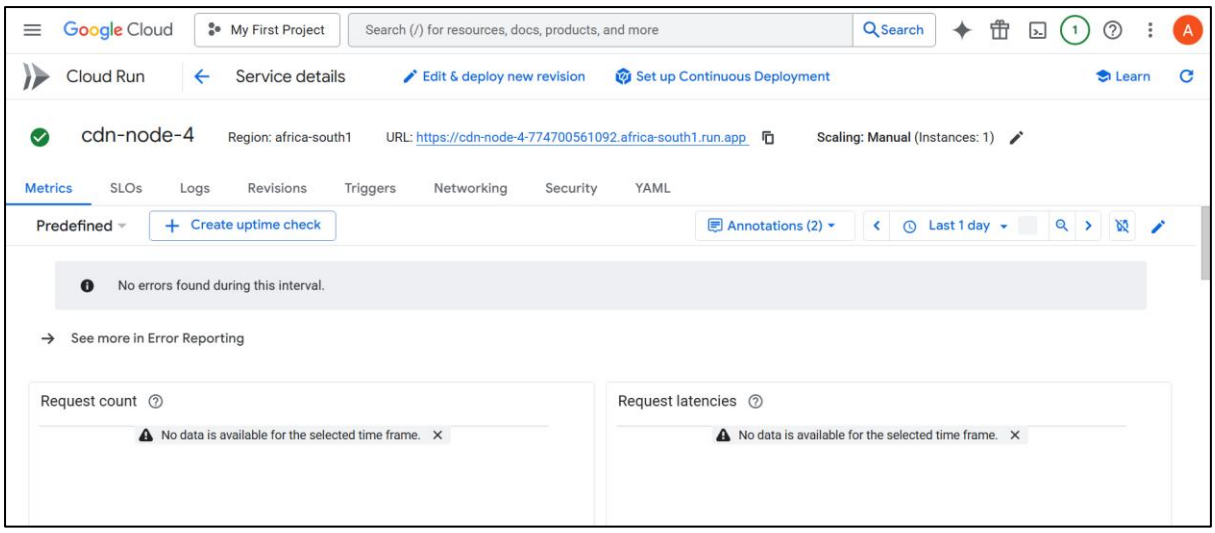
The Dashboard of the Node 1



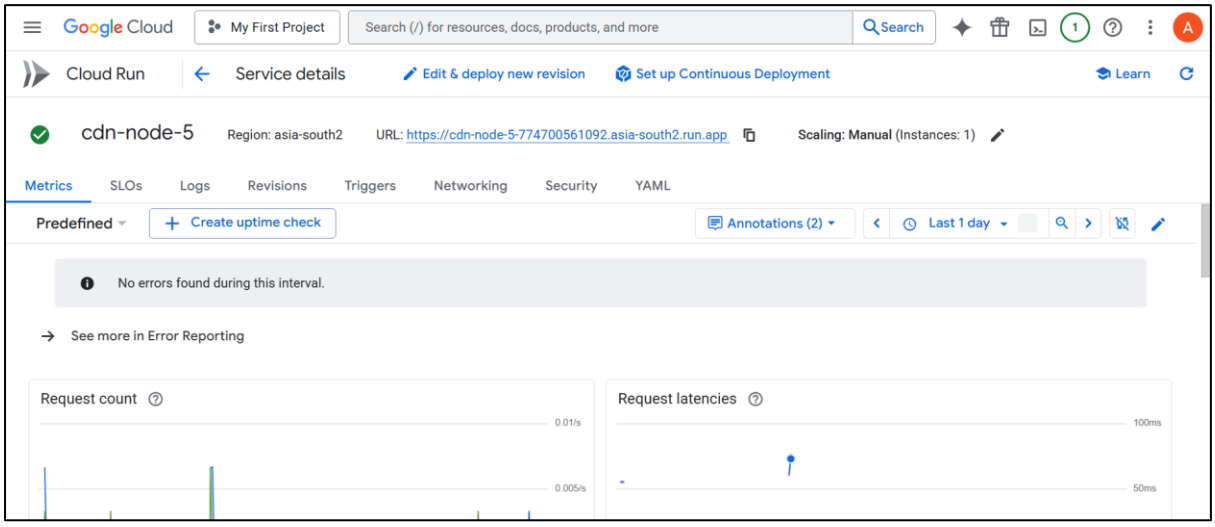
The Dashboard of the Node 2



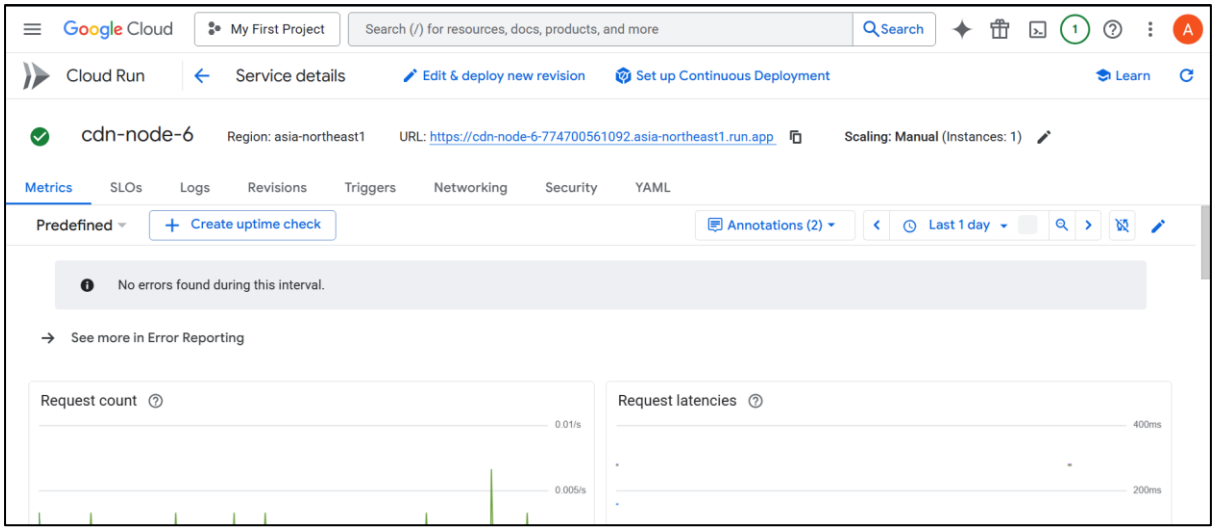
The Dashboard of the Node 3



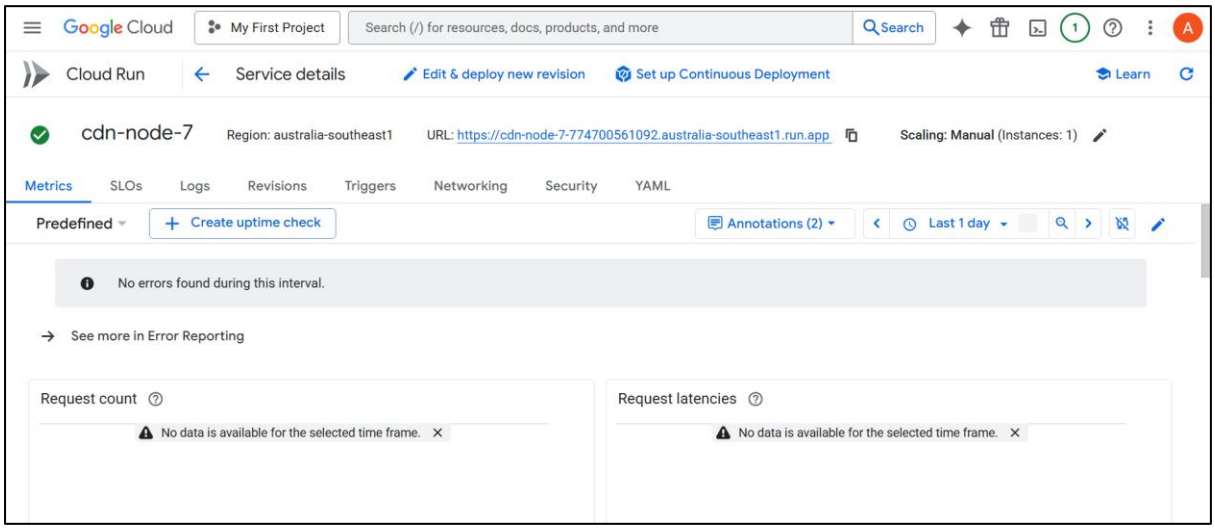
The Dashboard of the Node 4



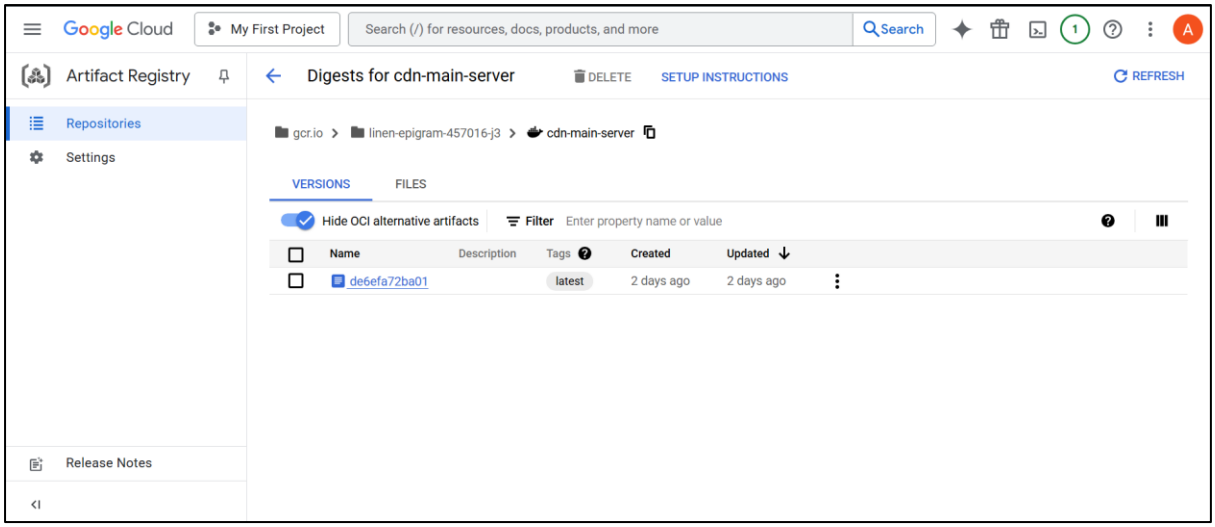
The Dashboard of the Node 5



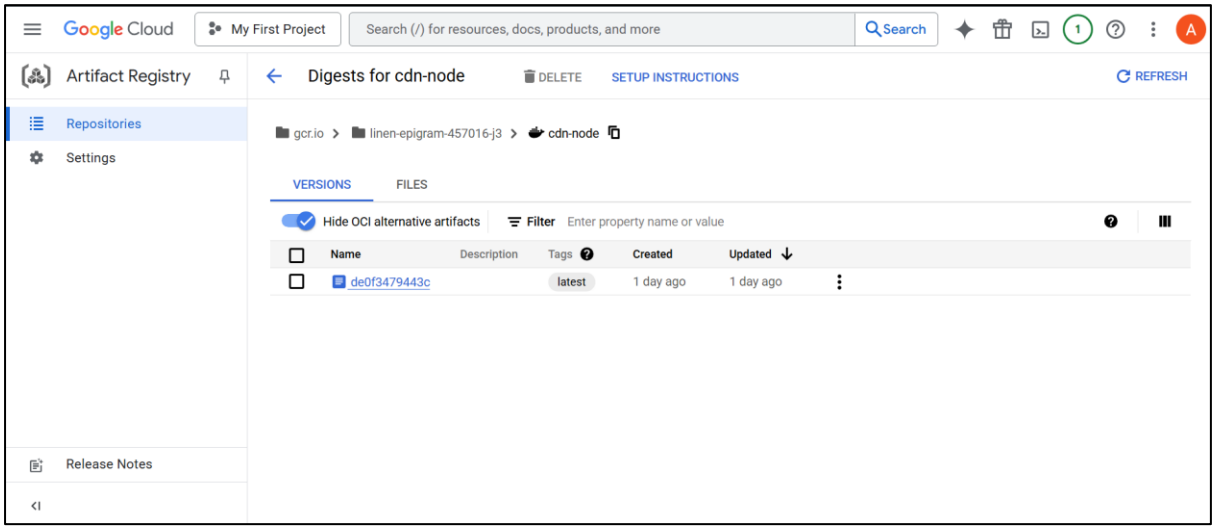
The Dashboard of the Node 6



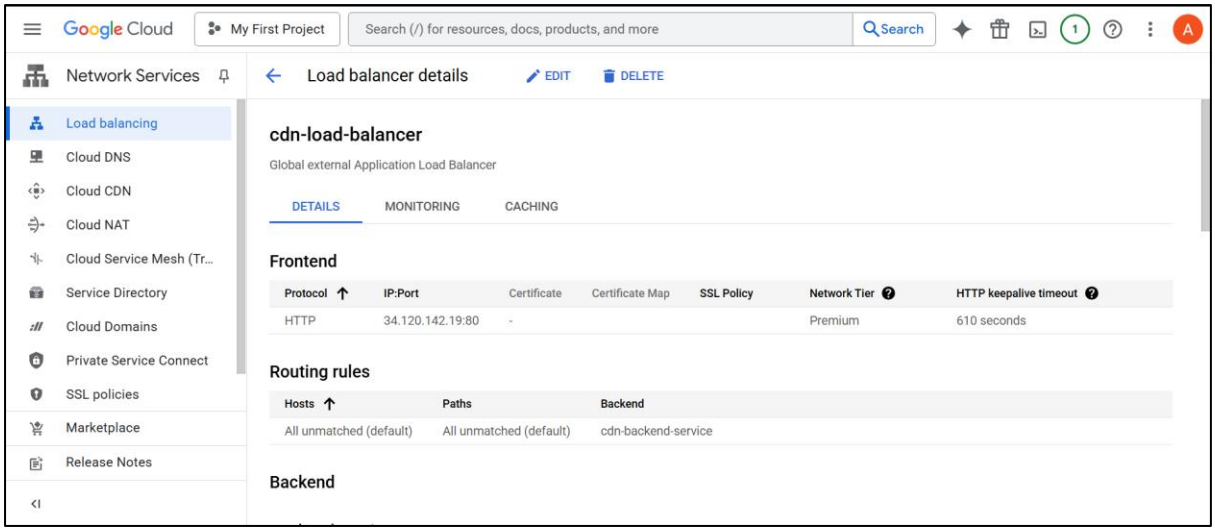
The Dashboard of the Node 7



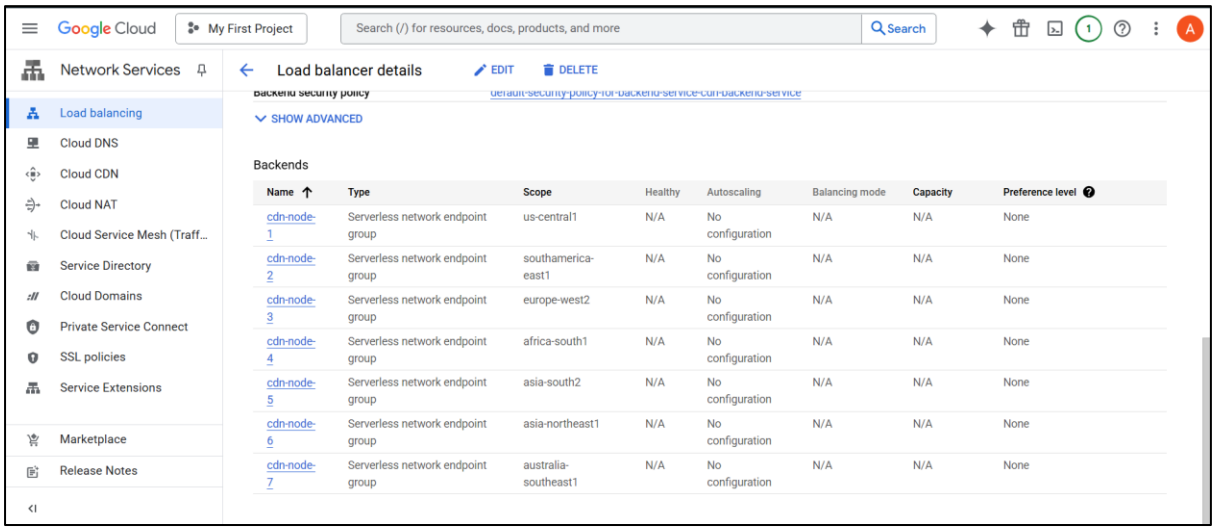
The Dashboard of the Artifact Registry showing docker image of server



The Dashboard of the Artifact Registry showing docker image of node



The Dashboard of the Load Balancer showing IP Address



The Dashboard of the Load Balancer showing the created Serverless NEGs and it's Locations

CONCLUSION:

The development and deployment of a pull-based Content Delivery Network (CDN) using Google Cloud Platform (GCP) represents a significant step toward understanding modern distributed systems and cloud-native architecture. This project demonstrates the power of combining stateless, event-driven microservices with managed infrastructure to create a scalable, low-latency, and cost-effective content delivery solution.

At the core of the system is a modular architecture, composed of a **main server** deployed on **Google Kubernetes Engine (GKE)** and **seven edge nodes** deployed on **Cloud Run** across multiple geographic regions. The architecture supports dynamic content fetching, caching with deduplication, and global distribution via a single global IP address using **HTTP(S) Global Load Balancing**. Each edge node uses **Redis** for fast, in-memory caching and maintains **TTL-based expiration** to ensure content freshness.

One of the major strengths of this implementation is its use of **Redis Pub/Sub** for propagating cache updates and invalidations. This event-driven communication model ensures that changes made at the main server level are efficiently synchronized across all distributed edge nodes, achieving **eventual consistency** without tight coupling or polling overhead.

The CDN supports core content operations (GET, POST, DELETE) through RESTful HTTP APIs, built in **Go** for performance and concurrency efficiency. The content is stored in base64-encoded format, and deduplication is implemented through **SHA-512 hashing**, which maps each file to a unique hash stored in Redis. This design not only prevents redundant data storage but also simplifies content validation and referencing.

Performance testing was conducted using **Insomnia**, simulating client requests from different regions and observing response behavior. The results showed significant improvement in response times on cache hits, verifying the effectiveness of the caching mechanism and regional edge routing.

In terms of learning outcomes, this project offered deep insight into:

- Configuring and deploying containerized services using GKE and Cloud Run
- Designing stateless microservices with scalable communication
- Implementing caching strategies and cache coherency
- Managing global load balancing and secure service-to-service communication

The project successfully achieved its objective of demonstrating cloud computing expertise while also providing a working prototype of a real-world system that is scalable, secure, and extensible. It has laid a solid foundation for further enhancements and production-grade deployment.

FUTURE WORK:

While the current implementation serves as a solid proof-of-concept for a pull-based CDN using GCP services, there are several opportunities to enhance functionality, performance, flexibility, and observability in future iterations.

1. Support for All File Types

Currently, the system supports **text, JSON**, and **base64-encoded images**. In future versions, this can be extended to handle a broader range of file formats:

- Binary files (PDF, DOCX, EXE)
- Media files (MP4, WebM, MP3)
- Static website assets (CSS, JS, fonts) To achieve this, the server must support content-type headers and possibly use GCP's object storage (e.g., Cloud Storage) for heavy assets.

2. Intelligent Content Prefetching

Instead of purely relying on pull requests to populate edge caches, the system could be enhanced with **predictive prefetching algorithms**:

- Use analytics or historical access logs to determine high-demand content
- Periodically pre-populate caches before user demand arises
- Reduce first-time latency even further

3. Dynamic Edge Node Registration

Currently, the edge nodes are manually deployed and statically configured. A more production-like setup would include:

- Dynamic registration/deregistration of nodes
- Auto-scaling based on load or request frequency
- Integration with Kubernetes Service Discovery or Cloud Run metadata APIs

4. Observability and Monitoring

Add tools for **monitoring, alerting, and logging**:

- Integrate **Stackdriver Logging & Monitoring**
- Set up **Prometheus + Grafana** dashboards (if using GKE entirely)
- Log key metrics like:
    - Cache hit/miss ratio
    - Content request latency
    - Node load/availability

5. Cache Versioning and Rollback

Introduce **versioning** for cached content:

- Each cache key could be versioned (e.g., about.png:v2)
- Support rollback to older versions on request
- Allow time-based cache policies for rollback scenarios

6. Security Enhancements

While internal services are secure using VPC connectors, external-facing APIs could benefit from:

- OAuth 2.0 / API Key validation
- Role-based access to content (user/group scopes)
- JWT-based content access control (signed URLs)

7. Hybrid CDN Model with Cloud Storage Integration

Integrate GCP Cloud Storage for heavy or immutable content and serve it directly via signed URLs through edge nodes. This would:

- Offload heavy bandwidth from the main server
- Allow mixing of pull-based and object-based delivery in one CDN

CODE:

For main server:

1. server_main.go:

```
package main

import (
	"fmt"
	"net/http"
	"strings"
	"encoding/json"
	"log"
	"os"
	"os/signal"
	"syscall"
	"context"
)

// Pull load CDN
// GET (For fetching), POST (For adding new content), DELETE
// sudo service redis-server start

type RequestAndResponse struct {
	Value string `json:"value"`
}

const (
	updateChannel = "update-channel"
	deleteChannel = "delete-channel"
)

func response(w http.ResponseWriter, r *http.Request) {
	key := strings.TrimPrefix(r.URL.Path, "/")
	if key == "" {
		http.Error(w, "Missing the Key", http.StatusBadRequest)
		return
	}

	switch r.Method {
	case http.MethodGet:
		handleGet(w, key)
	case http.MethodPost:
		handlePost(w, r, key)
	case http.MethodDelete:
		handleDelete(w, key)
	default:
```

```
                http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        }
}

func handleGet(w http.ResponseWriter, key string)  {
        result, err := Get(key);
        if err != nil {
                http.Error(w, err.Error(), http.StatusNotFound)
                return
        }

        response := RequestAndResponse{Value: result}
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusOK)
        json.NewEncoder(w).Encode(response)
        fmt.Printf("GET - %d\n", http.StatusOK)
}

func handlePost(w http.ResponseWriter, r *http.Request, key string) {
   var data RequestAndResponse
   if err := json.NewDecoder(r.Body).Decode(&data); err != nil {
      http.Error(w, err.Error(), http.StatusBadRequest)
      return
   }
   defer r.Body.Close()

   if err := Set(key, data.Value); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
      return
         }

   w.WriteHeader(http.StatusOK)
        Publish(updateChannel, fmt.Sprintf("key:%s;value:%s", key, data.Value))
        fmt.Printf("POST - %d\n", http.StatusOK)
}

func handleDelete(w http.ResponseWriter, key string){
        if err := Delete(key); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
        }
        w.WriteHeader(http.StatusOK)
        Publish(deleteChannel, fmt.Sprintf("key:%s", key))
        fmt.Printf("DELETE - %d\n", http.StatusOK)
}

func main() {
        InitRedis()
        InitCentralRedis()

        port := os.Getenv("PORT")
```

```
	http.HandleFunc("/", response)
	srv := &http.Server{Addr: ":" + port}

	go func() {
		fmt.Println("Listening on port " + port)
		if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
			log.Fatalf("HTTP server failed: %s\n", err)
		}
	}()

	stop := make(chan os.Signal, 1)
	signal.Notify(stop, os.Interrupt, syscall.SIGTERM)

	<-stop
	fmt.Println("\nShutting down server...")

	ctx, cancel := context.WithTimeout(context.Background(), 5)
	defer cancel()

	if err := srv.Shutdown(ctx); err != nil {
		log.Fatalf("Server shutdown failed: %s\n", err)
	}

	fmt.Println("Server exited properly.")
}
```

2. server_redis.go:

```
package main

import(
	"context"
	"fmt"
	"github.com/redis/go-redis/v9"
	"crypto/sha512"
	"os"
)

var (
	ctx = context.Background()
	client *redis.Client
	centralClient *redis.Client
	redisAddr = os.Getenv("REDIS_ADDR")
	centralRedisAddr = os.Getenv("CENTRAL_REDIS_ADDR")
)

func GetHash(base64 string) string {
	hasher := sha512.New()
	hasher.Write([]byte(base64))
```

```
	return fmt.Sprintf("%x", hasher.Sum(nil))
}

func InitRedis() {
	client = redis.NewClient(&redis.Options{
		Addr: redisAddr,
		Password: "",
		DB: 0,
	})

	if _, err := client.Ping(ctx).Result(); err != nil {
		fmt.Println("Failed to connect to Redis: " + err.Error())
		return
	}

	fmt.Println("Connected to Redis")
}

func Get(key string) (string, error) {
	hash, err := client.Get(ctx, key).Result()
	if err != nil {
		return "", err
	}

	return client.Get(ctx, hash).Result()
}

func Set(key, value string) error {
	hash := GetHash(value)

	if _, err := client.Get(ctx, hash).Result(); err != nil {
		client.Set(ctx, hash, value, 0).Err()
	}

	return client.Set(ctx, key, hash, 0).Err()
}

func Delete(key string) error {
	hash, err := client.Get(ctx, key).Result()
	if err != nil {
		return err
	}

	if _, err := client.Del(ctx, key).Result(); err != nil {
		return err
	}

	count, err := client.Keys(ctx, "*").Result()
	if err != nil {
		return err
```

```
	}

	otherReferences := false
	for _, k := range count {
		h, _ := client.Get(ctx, k).Result()
		if h == hash {
			otherReferences = true
			break
		}
	}

	if !otherReferences {
		client.Del(ctx, hash)
	}

	return nil
}

func InitCentralRedis() {
	centralClient = redis.NewClient(&redis.Options{
		Addr: centralRedisAddr,
		Password: "",
		DB: 0,
	})

	if _, err := centralClient.Ping(ctx).Result(); err != nil {
		fmt.Println("Failed to connect to Redis: " + err.Error())
		return
	}

	fmt.Println("Connected to Central Redis")
}

func Publish(channel, message string) error {
	return centralClient.Publish(ctx, channel, message).Err()
}

func Subscribe(channel1, channel2 string) (*redis.PubSub, error) {
	pubsub := centralClient.Subscribe(ctx, channel1, channel2)
	if _, err := pubsub.Receive(ctx); err != nil {
		return nil, err
	}
	return pubsub, nil
}
```

3. dockerfile:

```
FROM golang:1.24 as builder

WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

COPY . .

RUN go build -o server server_main.go server_redis.go

FROM debian:bookworm-slim

RUN apt-get update && \
    apt-get install -y redis-server ca-certificates && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY --from=builder /app/server .
COPY start.sh .

RUN chmod +x start.sh

ENV REDIS_ADDR=localhost:6379
ENV CENTRAL_REDIS_ADDR=localhost:6380
ENV PORT=5000

EXPOSE 5000 6379 6380

ENTRYPOINT ["./start.sh"]
```

4. start.sh:

```
#!/bin/bash

mkdir -p /tmp/redis1 /tmp/redis2

redis-server --port 6379 --dir /tmp/redis1 --daemonize yes
redis-server --bind 0.0.0.0 --port 6380 --dir /tmp/redis2 --daemonize yes --protected-mode no

sleep 1

./server
```

For node:

1. node_main.go:

```
package main

import (
	"fmt"
	"net/http"
	"strings"
	"encoding/json"
	"log"
	"os"
	"os/signal"
	"syscall"
	"context"
)

const (
	updateChannel = "update-channel"
	deleteChannel = "delete-channel"
)

func response(w http.ResponseWriter, r *http.Request) {
	key := strings.TrimPrefix(r.URL.Path, "/")
	if key == "" {
		http.Error(w, "Missing the Key", http.StatusBadRequest)
		return
	}

	switch r.Method {
	case http.MethodGet:
		handleGet(w, key)
	case http.MethodPost:
		handlePost(w, r, key)
	case http.MethodDelete:
		handleDelete(w, key)
	default:
		http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
	}
}

func handleGet(w http.ResponseWriter, key string)  {
	result, err := NodeGet(key);
	if err != nil {
		http.Error(w, err.Error(), http.StatusNotFound)
		return
	}

	response := RequestAndResponse{Value: result}
```

```
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusOK)
        json.NewEncoder(w).Encode(response)
        fmt.Printf("GET - %d\n", http.StatusOK)
}

func handlePost(w http.ResponseWriter, r *http.Request, key string) {
   var data RequestAndResponse
   if err := json.NewDecoder(r.Body).Decode(&data); err != nil {
      http.Error(w, err.Error(), http.StatusBadRequest)
      return
   }
   defer r.Body.Close()

   if err := NodeSet(key, data.Value); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
      return
         }
   w.WriteHeader(http.StatusOK)
        fmt.Printf("POST - %d\n", http.StatusOK)
}

func handleDelete(w http.ResponseWriter, key string){
        if err := NodeDelete(key); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
        }
        w.WriteHeader(http.StatusOK)
        fmt.Printf("DELETE - %d\n", http.StatusOK)
}

func KeyExists(key string) (bool, error) {
   count, err := client.Exists(ctx, key).Result()
   if err != nil {
      return false, err
   }
   return count > 0, nil
}

func main() {
        InitRedis()
        InitCentralRedis()

        pubsub, err := Subscribe(updateChannel, deleteChannel)
        if err != nil {
                log.Fatal("Subsription to Pub/Sub Channel Failed")
                return
        }

        ch := pubsub.Channel()
        go func() {
```

```
		for msg := range ch {
			switch msg.Channel {
			case updateChannel:
				fmt.Println("Received a Update request from Pub/Sub
Channel")
				res := strings.Split(msg.Payload, ";")
				key := strings.TrimPrefix(res[0], "key:")
				value := strings.TrimPrefix(res[1], "value:")
				if exists, _ := KeyExists(key); exists {
					if err := Set(key, value); err != nil {
						log.Print(err)
					}
				}

			case deleteChannel:
				fmt.Println("Received a Delete request from Pub/Sub
Channel")
				key := strings.TrimPrefix(msg.Payload, "key:")
				if exists, _ := KeyExists(key); exists {
					if err := Delete(key); err != nil {
						log.Print(err)
					}
				}
			}
		}
	}()

	port := os.Getenv("PORT")
	http.HandleFunc("/", response)
	srv := &http.Server{Addr: ":" + port}

	go func() {
		fmt.Println("Listening on port " + port )
		if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
			log.Fatalf("HTTP server failed: %s\n", err)
		}
	}()

	stop := make(chan os.Signal, 1)
	signal.Notify(stop, os.Interrupt, syscall.SIGTERM)

	<-stop
	fmt.Println("\nShutting down node...")

	ctx, cancel := context.WithTimeout(context.Background(), 5)
	defer cancel()

	if err := srv.Shutdown(ctx); err != nil {
		log.Fatalf("Node shutdown failed: %s\n", err)
	}
```

```
	fmt.Println("Node exited properly.")
}
```

2. node_redis.go:

```
package main

import (
	"bytes"
	"context"
	"crypto/sha512"
	"encoding/json"
	"fmt"
	"io"
	"net/http"
	"os"
	"time"
	"github.com/redis/go-redis/v9"
)

var (
	ctx = context.Background()
	client *redis.Client
	centralClient *redis.Client
	serverURL = os.Getenv("SERVER_URL")
	redisAddr = os.Getenv("REDIS_ADDR")
	centralRedisAddr = os.Getenv("CENTRAL_REDIS_ADDR")
)

type RequestAndResponse struct {
	Value string `json:"value"`
}


func GetHash(base64 string) string {
	hasher := sha512.New()
	hasher.Write([]byte(base64))
	return fmt.Sprintf("%x", hasher.Sum(nil))
}

func InitRedis() {
	client = redis.NewClient(&redis.Options{
		Addr: redisAddr,
		Password: "",
		DB: 0,
	})
```

```
	if _, err := client.Ping(ctx).Result(); err != nil {
		fmt.Println("Failed to connect to Redis: " + err.Error())
		return
	}

	fmt.Println("Connected to Redis")
}

func Get(key string) (string, error) {
	hash, err := client.Get(ctx, key).Result()
	if err != nil {
		return "", err
	}

	return client.Get(ctx, hash).Result()
}

func Set(key, value string) error {
	hash := GetHash(value)

	if _, err := client.Get(ctx, hash).Result(); err != nil {
		client.Set(ctx, hash, value, 0).Err()
	}

	ttl := 24 * time.Hour
	return client.Set(ctx, key, hash, ttl).Err()
}

func Delete(key string) error {
	hash, err := client.Get(ctx, key).Result()
	if err != nil {
		return err
	}

	if _, err := client.Del(ctx, key).Result(); err != nil {
		return err
	}

	count, err := client.Keys(ctx, "*").Result()
	if err != nil {
		return err
	}

	otherReferences := false
	for _, k := range count {
		h, _ := client.Get(ctx, k).Result()
		if h == hash {
			otherReferences = true
			break
		}
```

```
	}

	if !otherReferences {
		client.Del(ctx, hash)
	}

	return nil
}


func NodeGet(key string) (string, error) {
	base64, err := Get(key)
	if err != nil {
		resp, err := http.Get(serverURL + key)
		if err != nil {
			return "", err
		}
		defer resp.Body.Close()

		body, err := io.ReadAll(resp.Body)
		if err != nil {
			return "", err
		}

		var result RequestAndResponse
		if err = json.Unmarshal(body, &result); err != nil {
			return "", err
		}

		if err = Set(key, result.Value); err != nil {
			return "", err
		}

		fmt.Println("Value fetched from Main Server")
		return result.Value, nil
	} else {
		return base64, nil
	}
}

func NodeSet(key, value string) error {
	if err := Set(key, value); err != nil {
		return err
	}

	body := RequestAndResponse{Value: value}
	payload, err := json.Marshal(body)
	if err != nil {
		return err
	}
```

```
	_, err = http.Post(serverURL + key, "application/json", bytes.NewReader(payload))
	return err
}

func NodeDelete(key string) error {
	if err := Delete(key); err != nil {
		return err
	}

	body := RequestAndResponse{Value: key}
	payload, err := json.Marshal(body)
	if err != nil {
		return err
	}

	request, err := http.NewRequest(http.MethodDelete, serverURL + key,
bytes.NewReader(payload))
	if err != nil {
		return err
	}

	client := &http.Client{}
	resp, err := client.Do(request)
	if err != nil {
		return err
	}

	defer resp.Body.Close()

	if resp.StatusCode != http.StatusOK {
		return fmt.Errorf("failed to delete, status code: %d", resp.StatusCode)
	}

	return nil
}

func InitCentralRedis() {
	centralClient = redis.NewClient(&redis.Options{
		Addr: centralRedisAddr,
		Password: "",
		DB: 0,
	})

	if _, err := centralClient.Ping(ctx).Result(); err != nil {
		fmt.Println("Failed to connect to Redis: " + err.Error())
		return
	}

	fmt.Println("Connected to Central Redis")
```

```
}

func Publish(channel, message string) error {
	return centralClient.Publish(ctx, channel, message).Err()
}

func Subscribe(channel1, channel2 string) (*redis.PubSub, error) {
	pubsub := centralClient.Subscribe(ctx, channel1, channel2)
	if _, err := pubsub.Receive(ctx); err != nil {
		return nil, err
	}
	return pubsub, nil
}
```

3. dockerfile:

```
# Build Stage
FROM golang:1.24 as builder

WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

COPY . .

RUN go build -o node node_main.go node_redis.go

FROM debian:bookworm-slim

RUN apt-get update && \
   apt-get install -y redis-server ca-certificates && \
   apt-get clean && \
   rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY --from=builder /app/node .
COPY start.sh .

RUN chmod +x start.sh

ENV REDIS_ADDR=localhost:6379
ENV CENTRAL_REDIS_ADDR=35.200.253.69:6380
ENV PORT=5000
ENV SERVER_URL=http://35.200.253.69:5000/

EXPOSE 5000 6379 6380
```

ENTRYPOINT ["./start.sh"]

4. start.sh:

```
#!/bin/bash

mkdir -p /tmp/redis1

redis-server --port 6379 --dir /tmp/redis1 --daemonize yes

sleep 1

./node
```