# Cloud Architecture and Optimization for a Distributed Application

**Guilherme Fernandes**
60045
gc.fernandes@campus.fct.unl.pt

**Francisco Moura**
60175
fte.moura@campus.fct.unl.pt

## Abstract

This project explores the migration of TuKano, a social media application for short-form video sharing, to Microsoft Azure's Cloud platform. The original version of TuKano is centralized and operates with in-memory data persistence, limiting its scalability, availability, and performance. This report details a migration effort, which reconfigures TuKano to take advantage of Azure's Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) offerings, including Azure Blob Storage, Azure Cosmos DB, and Azure Cache for Redis. The goal is to implement a modular, geographically adaptive architecture that can handle high traffic across regions, optimize data retrieval, and enhance overall scalability. Performance is evaluated across various configurations and geographic scopes, with an emphasis on the role of caching and database selection (NoSQL vs. SQL) effect on system throughput and latency. Findings highlight the benefits and challenges of transitioning a social media application to a cloud-native architecture, and provides insights into scaling for large, dispersed user bases.

## 1 Introduction

Cloud computing has transformed the architecture and deployment of modern web applications, enabling scalability, resource optimization, and high availability under varying load conditions. However, transitioning traditional, centralized applications to cloud-native solutions remains a complex challenge, particularly for social media platforms that demand efficient data handling and responsive user experiences.

This project addresses the challenge by migrating TuKano, a video-sharing social media network, to Microsoft Azure. The initial TuKano implementation is operated as a single-server application with tightly coupled services and in-memory data storage, limiting its scalability. This migration to Azure's PaaS services aims to preserve TuKano's core functionality while enhancing its scalability, availability, and regional adaptability.

The main objectives of this migration are: rearchitect TuKano's data management by adopting **Azure Blob Storage for media storage**, **Azure Cosmos DB for database management**, and **Azure Cache for Redis** to improve data retrieval. Second, it involves a performance evaluation comparing NoSQL and SQL backends on Azure Cosmos DB, assessing the impact of caching on system throughput and latency. Third, it implements a modular, global deployment strategy using Azure Traffic Manager, enabling optimal routing and failover across regions.

This report provides an overview of the architectural transformation, detailing the new modular, regionally adaptive design, performance insights, and the implications of cloud-based scalability for social media applications.

## 2 System Architecture

### 2.1 Architectural Overview

Our architected TuKano Cloud application, illustrated in Figure 1, incorporates a dual-region setup to ensure high availability and low latency, with a **primary** and a **secondary** Azure region.

Each region is equipped to support TuKano's functionalities, enabling seamless failover and balanced load distribution. Geo-replication ensures data redundancy, accessibility, and reduced latency for global users. However, as we prioritized core functionalities within time and budget constraints, the blob storage and PostgreSQL read/write operations cannot be done **within the**

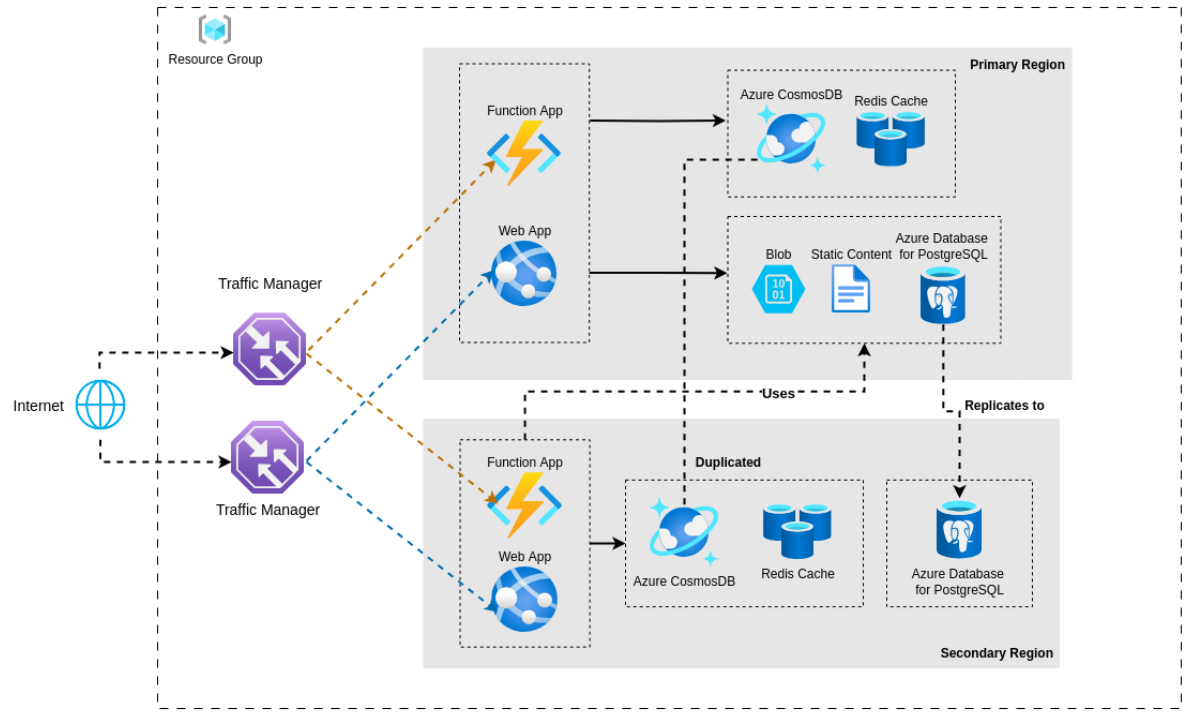


Figure 1: TuKano Azure Architecture

**secondary region**, as replication was limited to failover purposes, requiring access to the instances in the primary region for regular access, as shown in 1.

### 2.2 Component Descriptions

#### 2.2.1 Web App

The Web App component (depicted in Figure 1 within each region) hosts the core backend services for TuKano, managing requests related to user interactions, content feeds. The Web App in the primary region interacts directly with the other Azure resources (Cosmos DB, Redis Cache, PostgreSQL and Blob Storage) within the same region to minimize cross-region data transfer latency, while the Web App from the secondary region requires the Blob Storage and PostgreSQL from the primary region.

#### 2.2.2 Azure Cosmos DB (NoSQL and SQL Support)

TuKano leverages Azure Cosmos DB with both NoSQL and SQL (Cosmos DB for PostgreSQL) backends, each suited for different needs:

- **NoSQL:** This backend scales effectively through partitioning based on entities like user data, videos, and follow relationships. Updates are isolated within partitions, reducing conflicts and ensuring better performance. NoSQL uses the *etag* method to prevent race conditions and guarantees session consistency (Microsoft Transactions, 2024), ensuring that users see their own updates immediately. A retry mechanism further ensures data integrity during asynchronous operations, minimizing the risk of orphaned records.

  **Advantages:**

  - Highly scalable with partitioning, handling massive data volumes and concurrent interactions.
  - Flexible data model suited for real-time social interactions.
  - Optimized for low-latency reads and writes with session consistency.

  **Disadvantages:**

  - No immediate consistency ouside of a session, which can cause slight delays in data visibility.
  - Requires additional concurrency con-

trol, like etag checks, adding complexity.
  - Higher costs for cross-partition queries.

- **SQL (Cosmos DB for PostgreSQL):** This backend supports structured data, ideal for user profiles and relationships. It has a primary write node with a failover node for resilience. While offering strong consistency, its single-write node design can slowdown writes responses during high traffic scenarios, making it less optimal for write-heavy scenarios.

  **Advantages:**

  - Strong relational integrity and structured data support.
  - Familiar SQL interface for complex queries.
  - Built-in failover node for high availability.

  **Disadvantages:**

  - Single-write node limits scalability in write-heavy situations.
  - Failover introduces slight delays in data visibility.
  - Less optimized for handling unstructured data and real-time interactions.

TuKano's flexible architecture supports both NoSQL and SQL backends, enabling easy adaptation based on performance, scalability, or consistency needs. The NoSQL backend is ideal for fast, real-time interactions, while the SQL backend provides strong consistency for structured data, though with potentially slower writes under high load. This dual setup allows TuKano to optimize user experience as demands evolve.

### 2.2.3 Redis Cache

To balance performance with data consistency, TuKano integrates Azure Cache for Redis to optimize retrieval times and reduce load on the primary database. Our caching strategy focuses on selectively caching frequently accessed data, such as user feeds, follower lists, and item details, to handle high traffic effectively while keeping data up-to-date for users.

We employ targeted cache invalidation based on specific interactions like follows, likes, or content updates, ensuring that any cached data affected by these actions remains accurate. For example, if user A follows user B, we immediately invalidate the cached followers list for B to reflect the new follower count. This approach prevent data inconsistencies by ensuring that only frequently accessed data is cached.

**Time-To-Live (TTL) Management for Cache Keys**

Each cache entry in our system is configured with a Time-To-Live (TTL), defining how long it remains in the cache before expiring. This TTL mechanism enables a balance between data freshness and caching efficiency by adjusting durations according to the volatility and relevance of the data:

- **Short TTL for Dynamic Data:** Highly dynamic data, such as user feeds, is assigned a short TTL. This approach enables quick expiration, ensuring that frequently updated information remains current. User feeds, for example, may be refreshed multiple times by the same user within a short period. Caching feeds with a short TTL helps reduce database load from repeated requests while still reflecting recent updates promptly, thus balancing performance with data freshness.

- **Long TTL for Stable Data:** More stable data, like user profiles, content details, is given a longer TTL. By retaining this data in the cache longer, we minimize the load on the database without risking significant data inconsistency, as this information changes infrequently. Longer TTLs are particularly valuable for high-demand but stable data, as they reduce retrieval times and optimize cache resources.

TTL management is essential to cache integrity, ensuring that outdated data doesn't linger and that the cache focuses on freshness where it's most critical. With this dynamic TTL approach, our caching strategy minimizes unnecessary database load while maintaining low latency, offering users a responsive and current experience.

### 2.2.4 Azure Functions

Each region hosts a dedicated Azure Function App (at the top of each region block in Figure 1) to manage background tasks and event-driven functions, including the downloading and uploading of

blobs. We have implemented two specific functions for these tasks, anticipating a high volume of requests for both operations. The elastic nature of Azure Functions allows them to automatically scale in response to demand, ensuring efficient handling of concurrent requests without placing excessive load on the main application servers. Additionally, these functions are integrated with Azure Traffic Manager, which helps distribute incoming requests effectively across multiple function instances. This integration further enhances performance and reliability, allowing the application to maintain a seamless user experience during peak usage times.

### 2.3 Traffic Manager

For global traffic routing, we implemented Azure Traffic Manager, which not only optimizes load balancing across multiple regions but also ensures users are directed to the healthiest replicas. By monitoring the health of application endpoints, Traffic Manager intelligently routes requests to the available and most responsive region, reducing latency and ensuring high availability even during regional outages.

We had initially planned to use Azure Front Door for its advanced features like intelligent load balancing, SSL offloading, and endpoint routing. Specifically, the endpoint routing was intended to direct traffic to Azure Functions with minimal changes to the existing architecture. However, due to subscription limitations, we were unable to implement Azure Front Door, and instead, we relied on Azure Traffic Manager for global traffic distribution.

### 2.4 View Count Management

To efficiently manage view counts for shorts while minimizing database overload, we utilize atomic Redis counters. This approach allows us to increment view counts in the cache rapidly and reliably, which is essential given the expected high volume of view updates. When a user views a short, our system first checks if a counter for that specific short exists in the cache. If it does, we increment the counter directly. If the counter does not exist in the cache, we retrieve the current view count from the database, store it in Redis, and also store the timestamp of the last update in the cache to track when the view counter was last synchronized with the database. The cached timestamp of the last database synchronization helps us know when the cached count was last committed to the database, which is crucial for setting sync intervals and preventing data drift. This timestamp-driven sync strategy strikes a balance between performance and data precision. Instead of aiming for strict consistency, it ensures that view counts remain reasonably accurate without overwhelming the database.

We continuously monitor the time elapsed since the last update to the database. If the difference exceeds a predetermined threshold, indicating that the cached data may be stale, the update operation is triggered to synchronize the cached view count with the database without impacting the user experience.

While we aim to minimize the loss of view counts, we acknowledge that some updates may be missed during high traffic periods. Since the exact count of views is not critical to the application's functionality, minor discrepancies are acceptable. However, our strategy focuses on reducing this potential loss to maintain a balance between performance and data accuracy.

## 3 Non-Blocking Updates

To enhance the user experience, cache maintenance, including refreshes, invalidations, and periodic database syncs are executed asynchronously in separate threads. By decoupling these operations from the main application flow, we ensure that cache management happens seamlessly in the background, preserving front-end responsiveness and preventing delays.

## 4 Token Management and Modifications

### 4.1 Token Structure and Modifications

To optimize access control and session management, we transitioned to using JSON Web Tokens (JWTs) for our token structure. This change allows us to seamless validate the same token across multiple nodes, facilitating authentication regardless of which regional endpoint a user connects to. For every operation requiring authentication, users must send the JWT cookie.

## 5 Comparative Analysis of Components

To evaluate the impact of our architectural decisions, we conducted a series of performance tests focusing on database selection, caching, and geo-replication. Each comparison is based on empirical data gathered under typical application loads.

## 6 Specification of Resources for the tests

For the tests we used the following: France Central and Canada Central for the primary and secondary regions. Initially we wanted to use an US based secondary region, but due to multiple availability issues in all the US regions, we ended up using Canada.

Resources specifications:

- Azure Functions
  - Plan: Consumption
- Azure App Service
  - SKU: S3 – 4vCPUs and 7GB RAM
- Azure Cache for Redis
  - SKU: Basic
  - VM Size: C0
- Azure Blob Storage
  - Performance: Standard
  - Kind: StorageV2
  - Redundancy:
    - Read-access geo-redundant storage (RA-GRS)
    - Primary: France Central, Secondary: France South
- Azure CosmosDB for PostgreSQL
  - Coordinator Server Edition: General Purpose
  - Node Server Edition: Memory Optimized
- Azure CosmosDB for NoSQL
  - Consistency Policy: Session
  - Type: Standard
  - Read/Write Locations: France Central, Canada Central
- Traffic Manager
  - Routing Method: Performance

The test simulates a load test on a server endpoint that retrieves the user's feed. In this scenario, a single user performs 100 requests per second for 20 seconds to retrieve a feed of shorts, following 200 other users, each with at least one short. This dummy case is designed solely to assess the database performance, scalability, and response times under high traffic conditions.

### 6.1 NoSQL (Cosmos DB) vs. SQL (PostgreSQL) Performance

| Database Type | Response Time median (ms) | |
|---|---|---|
| | With Cache | Without Cache |
| NoSQL | 727 | 982 |
| SQL | 2328 | 4867 |

Table 1: Comparison of median requests response times with and without cache for NoSQL and SQL databases.

The performance comparison between SQL (PostgreSQL) and NoSQL (Cosmos DB) databases, specifically focusing on response times without caching, reveals notable differences. As shown in Table 1, SQL exhibits a significantly higher response time compared to NoSQL when caching is not used.

In the absence of caching, SQL's response time is 4867 milliseconds, while NoSQL's response time is 982 milliseconds. This represents a performance difference where Cosmos DB is approximately 5 times faster than PostgreSQL.

The slower performance of SQL can be attributed to its relational architecture, which requires more complex query processing. SQL enforces a rigid schema and ensures data integrity through operations like joins, which can increase the time it takes to execute queries, especially as the database scales. In contrast, NoSQL structure allows for faster retrieval due to its flexible, schema-less design, which avoids the need for such complex relational operations. This architecture is more efficient when querying large, distributed datasets, especially in scenarios that do not benefit from transactional consistency or relational data structures.

Thus, without caching, NoSQL databases like Cosmos DB demonstrate superior performance in terms of response time compared to SQL databases like PostgreSQL, which tend to experience higher latency due to the overhead associated with relational data management.

### 6.2 Caching vs. No Caching with Redis

Caching can greatly enhance the performance of both SQL (PostgreSQL) and NoSQL (Cosmos DB) databases by reducing the latency involved in data retrieval. Table 1 shows the response times for both types of databases with and without Redis caching.

With Redis caching, the response time for Cosmos DB (NoSQL) drops from 982 milliseconds to 727 milliseconds. Similarly, PostgreSQL (SQL) sees a reduction from 4867 milliseconds to 2328 milliseconds when Redis caching is enabled. These improvements indicate that Redis can accelerate response times by minimizing the need for repeated data retrieval directly from the database.

In SQL, caching has a more substantial decrease in response time compared to NoSQL. SQL queries often involve complex relational operations, which increase execution time. By storing frequently requested data in memory, Redis caching bypasses these processes, leading to a notable performance improvement.

In contrast, NoSQL is optimized for fast data retrieval due to their schema-less structure, so their uncached response times are already low. As a result, while caching still enhances NoSQL performance, the relative decrease in response time is generally smaller than in SQL database.

Thus, Redis caching benefits both SQL and NoSQL databases but has a more pronounced impact on SQL due to its higher baseline processing overhead.

### 6.3 Geo-Replication Performance Impact

#### 6.3.1 Access Speed and Latency Reduction

| **Response Time median (ms)** | |
|---|---|
| **Without Geo-Replication** | **With Geo-Replication** |
| 135 | 37 |

Table 2: Comparison of median requests response times with and without geo-replication for accessing the original app hosted in France from Canada, highlighting the impact of having a replica server closer to the user.

The performance comparison between access speeds with and without geo-replication, as shown in Table 2, highlights the significant reduction in response times when a geo-replicated server is used.

Without geo-replication, the response time is 135 milliseconds, indicating the time it takes to access the original server hosted in France from Canada. This relatively higher latency is expected due to the physical distance between the client and the server, resulting in longer round-trip times for data transmission.

However, when geo-replication is enabled, with a replica server placed closer to the user, the response time drops to just 37 milliseconds. This substantial reduction in latency is due to the fact that data is now being served from a server located geographically closer to the user, minimizing the time it takes for requests to travel between the client and the server. Geo-replication significantly improves access speed, especially for users located far from the original data center, by reducing network delays and enhancing overall application performance.

In conclusion, geo-replication proves to be highly effective in improving access speeds and reducing latency, offering a substantial performance boost for users accessing the app from remote locations.

#### 6.3.2 Availability and Reliability

Geo-replication also enhanced system resilience by enabling automatic failover in the event of regional outages. To test this, we deliberately ”killed” one of the replica servers and observed whether the application remained accessible. The result confirmed that users could still access the application, demonstrating the effectiveness of geo-replication in maintaining availability even when one replica server becomes unavailable.

While geo-replication improves availability and resilience, users located far from the remaining replicas may experience an increase in response time. This is due to the application relying on a more distant server to serve data, which introduces higher latency. However, the system’s ability to remain operational even during server or regional failures is a significant advantage in ensuring uninterrupted service.

## 7 Server Load Test and Response Time Analysis

To assess the server’s scalability and performance under various load conditions, we simulated a realistic usage scenario where different types of requests were made to the server. The test focused on measuring the server’s response time as the traffic volume (requests per second) increased. The results shown in Figure 7 represent the median response time over a 20-second period for each request rate.

As shown in the plot, there is a clear upward trend in response time as the number of requests

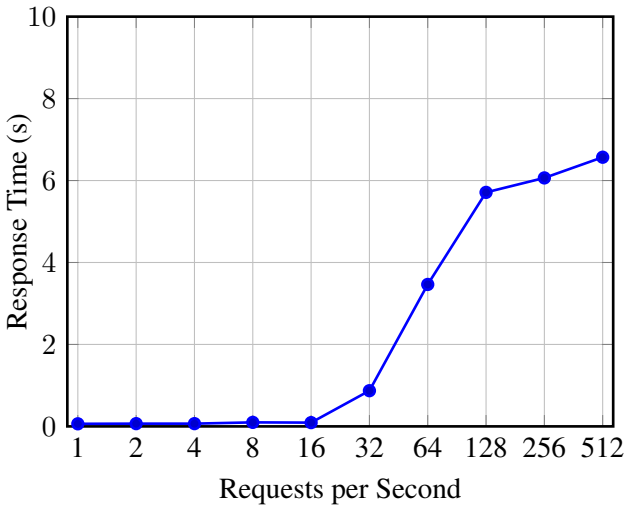


Figure 2: Requests response time (in seconds) as a function of requests per second. The chart shows how the response time increases as the request rate increases.

per second increases. At lower traffic volumes (e.g., 1 to 16 requests per second), the server responds relatively quickly, with response times remaining below 0.1 seconds. However, as the load increases beyond 32 requests per second, there is a noticeable increase in response time, with values rising above 0.5 seconds. At higher request rates (e.g., 128, 256, and 512 requests per second), the response time continues to escalate significantly, reaching over 6 seconds at 512 requests per second. This sharp rise in latency indicates that the server is beginning to struggle under the increasing load, suggesting potential bottlenecks in the system, such as limitations in processing power or bandwidth.

These results are essential for understanding the server's capacity and identifying the traffic threshold at which performance starts to degrade. By analyzing the data, we can make informed decisions on optimizing server infrastructure, such as scaling horizontally or fine-tuning the application to better handle increased traffic volumes.

## 8 Conclusion

This project highlights the benefits of migrating TuKano, a social media application, to a cloud-native architecture using Microsoft Azure's Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) offerings. By leveraging Azure Blob Storage, Azure Cosmos DB, and Azure Cache for Redis, the system's scalability, availability, and performance were significantly enhanced.

The integration of Azure's cloud services improved the application's ability to manage high traffic loads, reduce latency, and ensure resilience through geo-replication and high availability configurations. With a modular architecture, the application now benefits from efficient load distribution and seamless failover across multiple regions, optimizing performance for users worldwide. The caching mechanism provided by Redis alleviated database load, boosting response times for frequently accessed data.

The combination of Azure's Cosmos DB, offering both NoSQL and SQL capabilities, allowed the application to manage diverse data requirements with flexibility, ensuring robust performance even as usage expanded.

Ultimately, these architectural improvements enabled TuKano to scale more effectively, accommodate growing user demands, and maintain high performance.

## References

Microsoft. 2024. *Multi-Region Applications*. Retrieved from https://learn.microsoft.com/en-us/azure/architecture/web-apps/app-service/architectures/multi-region.

Microsoft. 2024. *Transactions and optimistic concurrency control*. Retrieved from https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/database-transactions-optimistic-concurrency.