*Article*

# Resilient Real-Time Data Delivery for AI Summarization in Conversational Platforms: Ensuring Low Latency, High Availability, and Disaster Recovery

**Shinoy Vengaramkode Bhaskaran** [1]

[1] Senior Big Data Engineering Manager, Zoom Video Communications.

**Abstract:** As conversational artificial intelligence (AI) agents become integral components of communication platforms the need for reliable and timely data delivery to AI summarization engines is paramount. This is true in the case of many domains from real-time customer support to interactive tutoring systems. Ensuring that conversational transcripts, user queries, and contextual metadata flow to summarization models with minimal latency and high fault tolerance is critical to maintaining seamless user experiences. This paper presents a comprehensive system design and technical strategies for achieving resilient real-time data delivery. We focus on architectural principles, low-latency data pipelines, fault-tolerant components, and disaster recovery mechanisms. By combining scalable streaming frameworks, distributed consensus protocols, geo-redundant storage, and active-active failover techniques, we demonstrate that it is feasible to maintain continuous availability, even in the face of network partitions and data center outages. Experimental evaluations on a prototypical testbed show our approach can maintain sub-100ms latency targets, minimize downtime under failure scenarios, and recover state swiftly and accurately.

**Keywords:** active-active failover, conversational AI, fault tolerance, low-latency pipelines, real-time data delivery, resilient system design, summarization engines

## 1. Introduction

Conversational AI platforms have become an integral part of modern communication, finding applications in diverse domains such as customer service chatbots, voice-based virtual assistants, and collaborative business tools. These platforms rely heavily on advanced natural language processing (NLP) models to analyze user inputs and provide concise, contextually accurate summaries of discussions in real-time. The ability of these summarization engines to deliver high-quality and relevant summaries is directly influenced by the performance, scalability, and fault tolerance of the underlying data delivery and processing infrastructure [1].

In static or offline settings, summarization engines can afford delays in processing data since there is no immediate user expectation for feedback. However, the situation is vastly different for interactive and real-time conversational systems. Users engaging with these systems expect instantaneous feedback and continuously updated insights as they generate substantial volumes of textual, audio, or multimodal data. Meeting these stringent latency and responsiveness requirements demands a robust architecture that can efficiently handle data ingestion, stream processing, summarization, and delivery, all while ensuring fault tolerance and minimal downtime [1,2].

Failures or disruptions in this infrastructure—whether caused by hardware malfunctions, software bugs, network congestion, or regional outages—can severely impact the system's reliability. Such interruptions do not merely degrade the user experience but also undermine user trust, as they lead to noticeable delays or inaccuracies in generated

**Citation:** Bhaskaran, S. V. Resilient Real-Time Data Delivery for AI Summarization in Conversational Platforms: Ensuring Low Latency, High Availability, and Disaster Recovery. *JICET* **2023**, *8*, 113–130.

Received: 203-06-18
Revised: 2023-08-08
Accepted: 2023-09-02
Published: 2023-09-12

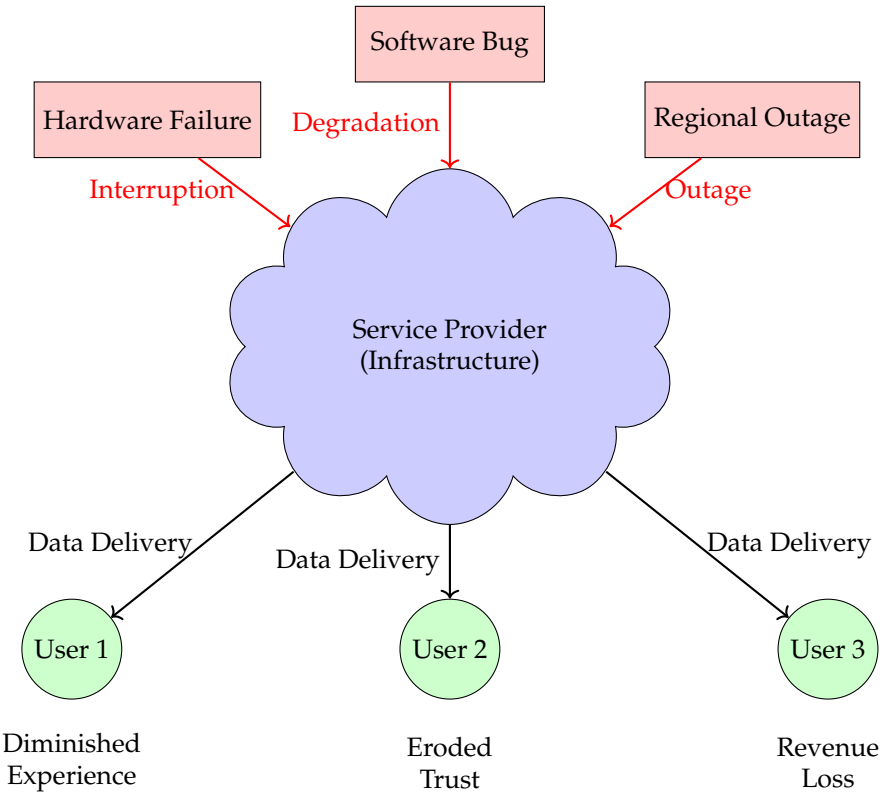


**Figure 1.** Visualization of the impact of interruptions or degradation in data delivery caused by hardware failures, software bugs, or regional outages. The resulting issues directly affect user experience, trust, and revenue.

summaries. For service providers, these performance lapses can translate to significant revenue losses, particularly in competitive markets where users have high expectations for reliability and responsiveness.

The challenges are further exacerbated by the growing scale and complexity of conversational data streams. The ingestion pipelines must accommodate high-throughput data flows while preserving data order and integrity. Streaming platforms and summarization engines must balance scalability with low-latency performance, even under sudden traffic spikes or infrastructure failures. To meet these demands, modern architectures incorporate cutting-edge technologies such as distributed streaming platforms, transformer-based language models, containerized microservices, and automated orchestration systems.

## 2. Challenges in Real-Time Summarization

Attaining real-time and resilient data delivery for AI summarization in distributed conversational systems presents a multitude of technical challenges. These challenges stem from the inherent complexity of processing large volumes of data at low latency while maintaining high availability, scalability, and fault tolerance. This section identifies the key hurdles that must be addressed to enable robust and efficient summarization in real-time settings [3,4].

A primary challenge is meeting low latency requirements, which demand that data ingestion, processing, and delivery to summarization models occur within tens of milliseconds. The strict temporal constraints arise from user expectations of immediate feedback, particularly in interactive settings. Achieving this necessitates overcoming network overhead, serialization costs, and queuing delays, all of which contribute to latency. Inefficiencies in any part of the pipeline can cascade, leading to delays that compromise the user experience [5–7].

Another significant hurdle is ensuring high availability and fault tolerance. Distributed systems are prone to various types of failures, ranging from individual server crashes to entire data center outages. For conversational AI platforms, even brief interruptions in the data pipeline can degrade system performance and lead to missing or incomplete summaries [8,9]. Consequently, maintaining continuous operation requires a fault-tolerant architecture with redundant components, replication strategies, and mechanisms for seamless failover. This ensures that service interruptions are minimized, even in the face of hardware or software failures.

Disaster recovery poses additional complexities, particularly in the context of large-scale catastrophic events such as natural disasters or network-wide outages [10,11]. In these scenarios, both data integrity and service continuity are at risk. Effective disaster recovery strategies must involve advanced replication mechanisms, geographically distributed backups, and processes for rapid failover and recovery. Ensuring data durability and correctness under such conditions is paramount, as these factors directly affect the quality of summarization outputs.

The need for scalability and elasticity further complicates system design. Conversational platforms often face unpredictable spikes in traffic, such as during major events or promotional campaigns [7,8,12]. Handling these dynamic workloads without compromising performance requires architectures that can scale elastically, provisioning additional resources during peak loads and scaling down during periods of reduced demand. Designing such adaptive systems involves significant challenges in resource allocation and management [13,14].

This paper addresses these challenges by proposing a resilient and low-latency data delivery architecture tailored for AI summarization in conversational platforms. Our primary contributions are as follows:

1. A distributed system design that integrates data ingestion, streaming frameworks, and AI inference engines, ensuring sub-100ms latency under nominal loads.
2. Techniques for achieving high availability through active-active deployments, load balancing, and distributed consensus mechanisms to sustain uninterrupted operations.
3. Disaster recovery strategies that incorporate continuous backup, geo-redundancy, and rapid failover mechanisms to minimize downtime and mitigate data loss.
4. An evaluation that demonstrates the efficacy of the proposed architecture using a representative testbed and synthetic workloads, highlighting key performance metrics.

The remainder of this paper is organized as follows: Section II outlines the proposed system architecture, detailing its key components, including data ingestion, message brokering, and integration with summarization engines. Section III focuses on strategies for latency optimization, addressing challenges in minimizing delays across the data pipeline. Section IV explores mechanisms for ensuring high availability, fault tolerance, and disaster recovery, providing a robust foundation for resilient operation. Section V presents experimental results and performance benchmarks, demonstrating the scalability and effectiveness of the proposed design.

## 3. Proposed System Architecture

The proposed system architecture is designed to manage the complexities associated with collecting, processing, and summarizing large volumes of conversational data in real-time. It integrates several subsystems, each dedicated to a specific functionality such as data ingestion, stream processing, summarization, and efficient load balancing. The architecture is both modular and scalable, ensuring robustness under variable workloads. The following subsections describe the individual components in detail.

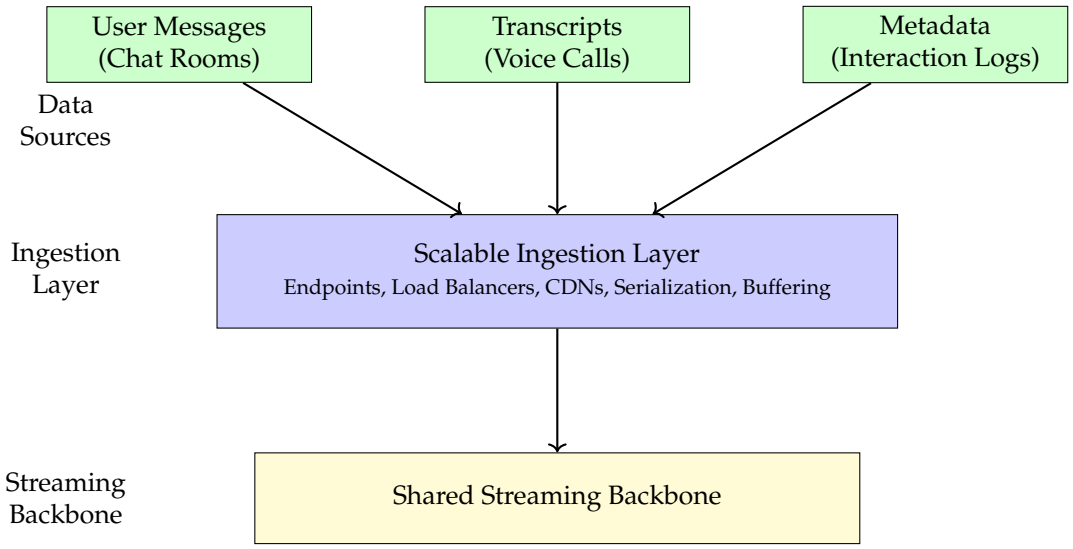


**Figure 2.** Data Ingestion Layer: Raw conversational data is collected from diverse sources such as chat rooms, voice call transcripts, and interaction logs. Ingestion endpoints, strategically deployed near user locations, utilize load balancers and CDNs to minimize latency. Messages are serialized and buffered before being forwarded to a shared streaming backbone for further processing.

### *3.1. Data Ingestion Layer*

The data ingestion layer serves as the initial entry point for collecting raw conversational data from heterogeneous sources. These sources include user-generated messages in chat rooms, voice call transcripts processed through speech-to-text systems, and metadata logs from customer interaction systems. The ingestion layer employs geographically distributed endpoints to minimize latency. These endpoints are typically deployed at network edges and are backed by Content Delivery Networks (CDNs) and load balancers, ensuring that data is routed to the nearest available ingestion node.

Each ingestion node is equipped with capabilities to serialize and buffer incoming data streams. The serialization ensures consistent structuring of data, which is critical for downstream processing, while buffering accommodates temporary spikes in data traffic. Data packets from ingestion nodes are subsequently forwarded to a central streaming backbone for distributed processing. The system's design emphasizes fault tolerance, with failover mechanisms in place to reroute data if any endpoint or ingestion node fails. Moreover, the ingestion layer supports a range of data protocols, including REST APIs, WebSocket streams, and gRPC interfaces, to ensure compatibility with diverse data sources.

### *3.2. Message Queues and Stream Processing*

At the core of the architecture lies the stream processing layer, which relies on high-performance distributed platforms such as Apache Kafka or Redpanda. These platforms provide an event-driven backbone that ensures data ordering, durability, and fault tolerance. Raw conversational data received from the ingestion layer is published to Kafka topics, with each topic corresponding to a specific type of conversation stream or metadata category.

Partitioning is employed within the stream processing layer to enhance scalability. Each user session or conversation is assigned to a unique partition, ensuring message order is preserved within individual sessions while enabling parallel processing across partitions. Partitions are distributed across multiple brokers within the Kafka cluster and are replicated to maintain high availability in the event of broker failures. For example, a replication factor of three ensures that a partition's data is available even if two brokers fail simultaneously.

Producers, typically the ingestion nodes, write data into Kafka topics, while consumer microservices subscribe to these topics for real-time processing. The consumer applications

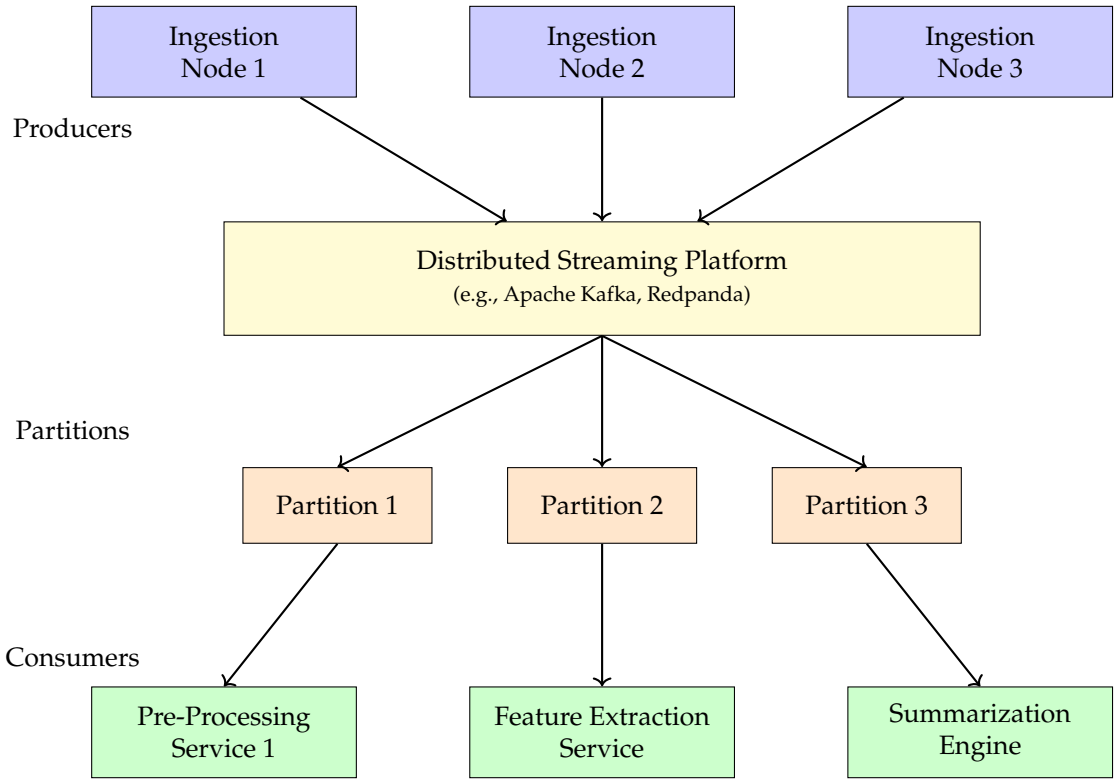


**Figure 3.** Message Queues and Stream Processing: ingestion nodes, write messages to partitions within a distributed streaming platform like Apache Kafka or Redpanda. Partitioning ensures throughput and ordering per conversation or user session. Partitions are replicated across a cluster for resilience. Consumers, including pre-processing services and summarization engines, subscribe to streaming topics, performing lightweight transformations before data moves forward.

perform a variety of pre-processing tasks, including feature extraction (e.g., sentiment scores or named entity recognition), normalization of textual data, and filtering of redundant or irrelevant information. These transformations are lightweight to ensure minimal processing delay, enabling near real-time data transmission to the downstream summarization engines.

### *3.3. Summarization Engine Integration*

The summarization engine is the most computationally intensive component of the architecture. It is implemented as a microservices-based system that transforms processed conversational data into concise summaries. These microservices are powered by state-of-the-art transformer-based models, such as GPT-style large language models (LLMs) or domain-specific pre-trained architectures like BERT. Given the complexity of these models, the summarization microservices require access to hardware accelerators such as GPUs, TPUs, or specialized AI inference chips.

To ensure scalability, the summarization services are containerized and deployed in a Kubernetes-managed environment. This orchestration allows dynamic scaling of resources in response to workload fluctuations. For instance, during peak traffic periods, additional instances of the summarization microservices can be spun up to handle the increased load. The microservices are designed to be stateless, with stateful information (e.g., intermediate summaries) stored in an external in-memory database like Redis.

A multi-tiered caching mechanism is employed to reduce computational load and minimize latency. The first tier is an in-memory cache that stores frequently accessed conversation segments and partial summaries. This enables incremental updates, where only new conversational elements are processed instead of re-summarizing the entire conversation history. The second tier is a distributed cache, which retains completed summaries for faster retrieval by downstream services.

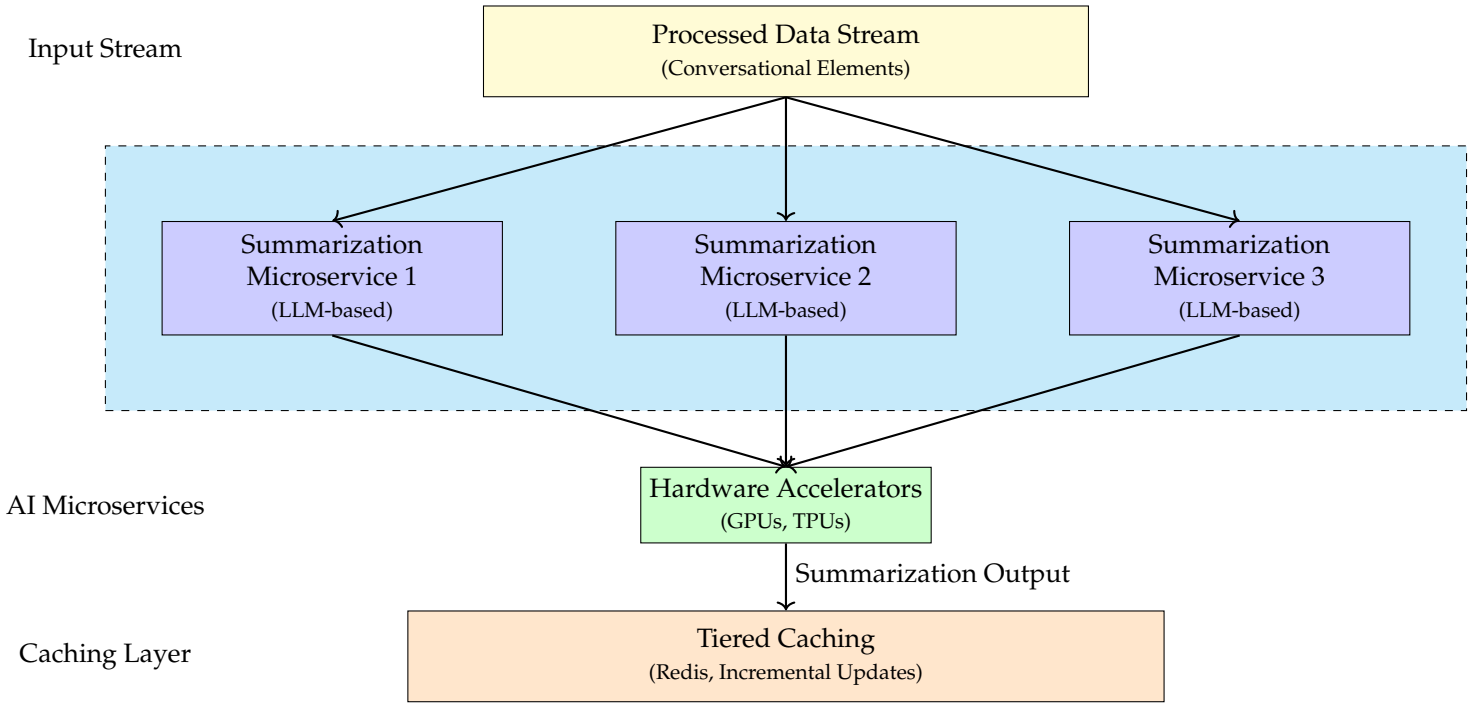


**Figure 4.** Summarization Engine Integration: Summarization engines, deployed as microservices, subscribe to processed data streams and transform conversational elements into concise summaries. These services use LLMs hosted in containerized environments managed by Kubernetes. To enhance performance and handle burst traffic, tiered caching is employed, with in-memory caches (e.g., Redis) storing recent conversation segments and incremental updates to minimize re-summarization.

**Table 1.** Technical Features of the Summarization Engine

| Feature | Description |
|---|---|
| Transformer-based Models | Utilizes large language models like GPT or BERT to generate concise summaries of conversational data. |
| Containerized Deployment | Services run in Docker containers managed by Kubernetes, ensuring scalability and fault tolerance. |
| Hardware Acceleration | Summarization engines leverage GPUs or TPUs for high-performance inference. |
| Multi-tier Caching | Combines in-memory and distributed caching layers to reduce redundant computations and improve latency. |

### *3.4. Caching and Load Balancing Components*

To manage high traffic loads and ensure uninterrupted service, the architecture integrates advanced caching and load balancing mechanisms. Load balancers, such as those provided by NGINX or HAProxy, route client requests to available summarization microservices based on a combination of criteria, including node health, proximity, and current workload. These load balancers perform real-time health checks on individual nodes, allowing them to dynamically redistribute traffic away from underperforming or failing instances.

Caching layers complement the load balancers by reducing the need for redundant computations. The caching system stores previously computed summaries and frequently requested data snippets, which are indexed for rapid retrieval. For example, if a client application requests a summary that has already been computed, the cache serves the response directly, bypassing the summarization engine entirely. Cache invalidation mechanisms

are carefully implemented to ensure that outdated summaries are refreshed only when underlying data changes.

Additionally, the system employs prewarming strategies to ensure that frequently accessed summaries remain in the cache, even during low-traffic periods. This strategy minimizes cache misses and reduces latency for users. Together, the caching and load balancing components provide a robust infrastructure for handling bursty traffic patterns while maintaining high performance.

**Table 2.** Key Functionalities of Caching and Load Balancing Components

| **Functionality** | **Description** |
|---|---|
| Dynamic Load Balancing | Routes requests to the most suitable summarization microservices, ensuring even distribution of workloads. |
| Real-Time Health Checks | Continuously monitors microservices and reroutes traffic away from failing nodes. |
| Efficient Caching | Stores previously computed summaries and frequently accessed data for fast retrieval. |
| Cache Prewarming | Proactively loads frequently requested summaries into the cache to reduce latency. |

### *3.5. End-to-End Scalability and Fault Tolerance*

The system architecture is designed with scalability and fault tolerance as primary considerations. Horizontal scaling is supported at all layers, from the ingestion nodes to the summarization microservices. Redundancy is built into every component, ensuring that the system can recover gracefully from hardware or network failures. For example, ingestion nodes and brokers are deployed in geographically distributed clusters, minimizing the impact of localized outages.

In addition, the system supports elastic scaling, where resources can be automatically added or removed based on traffic patterns. This is achieved through auto-scaling rules defined within the orchestration platform, allowing the system to adapt to sudden changes in demand without manual intervention. Fault detection and recovery mechanisms further enhance the system's reliability, enabling continuous operation even under adverse conditions.

This end-to-end architecture ensures a seamless, high-performance environment for processing and summarizing conversational data, meeting the demands of modern real-time applications.

## 4. Ensuring Low Latency

Low latency is a critical requirement for real-time AI summarization pipelines, particularly when handling large volumes of conversational data across distributed systems. Achieving this goal necessitates the optimization of multiple architectural layers, including data ingestion, transport protocols, serialization, and summarization workflows. This section deals with the strategies employed to ensure sub-second latency while maintaining high throughput and system resilience.

### *4.1. High-Throughput Ingestion Strategies*

A high-throughput ingestion layer is essential for minimizing queuing delays in real-time pipelines. One effective strategy involves the use of batching and micro-batching techniques. While a fully streaming architecture offers the lowest latency, processing data in micro-batches of just a few milliseconds can significantly reduce the overhead associated with network calls, serialization, and acknowledgments. For example, batching 5–10ms worth of incoming messages amortizes these costs, resulting in up to a 20% reduction in latency without introducing perceptible delays in the overall pipeline. Frameworks

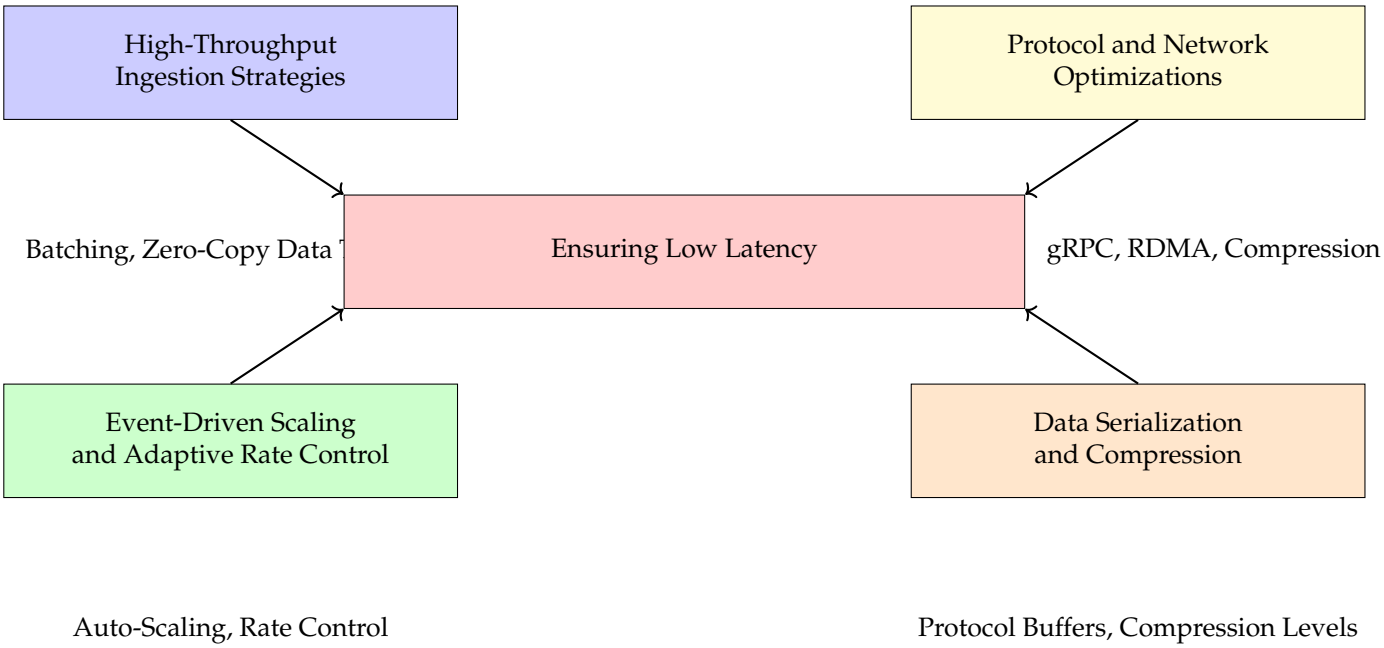


**Figure 5.** Ensuring Low Latency: The system employs multiple strategies to maintain low latency, including high-throughput ingestion with batching and zero-copy data transfer, optimized protocols and network infrastructure (e.g., gRPC, RDMA), event-driven scaling to handle fluctuating workloads, and efficient serialization and compression techniques tailored to current conditions. Together, these measures reduce delays while maximizing throughput and resilience.

designed for streaming, such as Kafka, naturally support this approach by batching writes to message queues and applying back-pressure to regulate downstream flow.

Additionally, zero-copy data transfer techniques reduce CPU overhead by eliminating intermediate memory copying steps. Traditional pipelines frequently copy data multiple times—from kernel to user space, from broker buffers to application-level memory, and so on. Modern systems mitigate this inefficiency by leveraging zero-copy protocols such as 'sendfile()' on Unix-like operating systems or RDMA (Remote Direct Memory Access) in advanced data center networks. In Kafka-based pipelines, for instance, data can be transferred directly from a broker's page cache to a consumer application without intermediate copies, shaving off critical milliseconds of latency.

To further enhance ingestion performance, edge-based data collection is employed. Ingestion nodes deployed at the network edge—close to the users generating conversational data—reduce WAN latency by processing data locally before forwarding it to the central pipeline. These edge nodes can perform lightweight operations such as data normalization, filtering, and compression, offloading some processing tasks from the core infrastructure. This localized preprocessing improves both responsiveness and scalability by reducing the volume of raw data transported to central servers.

### *4.2. Protocol and Network Optimizations*

The choice of transport protocols and network configurations plays a pivotal role in minimizing latency. Modern application-level protocols such as gRPC over HTTP/2 outperform older protocols by leveraging features like multiplexed streams, header compression, and built-in flow control. These advancements reduce the round-trip time for request-response cycles and ensure better utilization of network resources. Emerging QUIC-based protocols, which operate over UDP, further improve tail latency by reducing handshake overhead and enabling fast reconnections.

Hardware-accelerated networking techniques are also employed to reduce latency within data centers. RDMA, combined with kernel-bypass networking stacks like DPDK

(Data Plane Development Kit), enables direct memory access across nodes without CPU intervention, drastically reducing round-trip times. These optimizations are particularly effective in high-throughput environments, where microsecond-level improvements can have a significant cumulative impact.

Encryption overhead is another factor that affects latency. While data security is essential, TLS termination incurs computational costs. This challenge can be addressed by deploying TLS offloading hardware or using cryptographic accelerators. Selective encryption offers an additional solution by allowing less sensitive metadata to traverse lower-latency paths, while end-to-end encryption is maintained for critical conversation content. Intelligent compression strategies further optimize network usage. For instance, adaptive compression dynamically adjusts the compression level based on network conditions, employing fast algorithms like LZ4 during low-CPU, high-bandwidth scenarios and more aggressive methods like Zstandard under peak loads.

### *4.3. Event-Driven Scaling and Adaptive Rate Control*

To maintain low latency during sudden traffic surges or transient failures, the system relies on event-driven scaling and adaptive rate control mechanisms. Horizontal autoscaling of summarization microservices is a core strategy, with orchestration platforms such as Kubernetes monitoring key metrics like message backlog, GPU utilization, and latency. When thresholds are exceeded, additional containerized summarization instances are automatically provisioned, ensuring that the system can handle increased load. Conversely, instances are decommissioned during low-traffic periods to conserve resources.

Adaptive rate limiting is another crucial mechanism for managing congestion in the pipeline. When downstream components like summarization engines or storage tiers experience high latency, ingestion rates are dynamically throttled to prevent cascading backlogs. A credit-based flow control system adjusts the ingestion speed in real-time, prioritizing latency-sensitive tasks while temporarily deferring non-critical data.

Event-driven architectures enable rapid responses to anomalies such as spikes in latency or resource exhaustion. Load-shedding mechanisms are employed in extreme cases, where low-priority tasks are deferred or dropped to preserve performance for high-priority users. This selective degradation aligns with service-level objectives (SLOs) that prioritize premium users during peak demand, maintaining system reliability under stress.

### *4.4. Data Serialization and Compression Approaches*

Efficient serialization and compression are vital for reducing latency while minimizing resource consumption. Binary serialization formats such as Protocol Buffers, FlatBuffers, and Cap'n Proto offer significant advantages over text-based formats like JSON. These formats produce smaller payloads and require fewer CPU cycles for serialization and deserialization, leading to more predictable low-latency performance under heavy load.

Schema evolution and forward compatibility are also critical considerations. By employing schema-aware serialization methods, the system can evolve its data models without breaking compatibility. Precompiled schemas eliminate costly runtime lookups or dynamic parsing, ensuring that decoding operations remain efficient.

Selective compression strategies further optimize data transmission. For example, large text fields or high-entropy binary data are compressed using lightweight algorithms like Snappy or LZ4, which offer high-speed compression with moderate size reductions. For data requiring greater compression, Zstandard can be used in balanced mode, providing better ratios while keeping decompression times low. By tailoring compression approaches to specific data types, the system achieves a balance between bandwidth savings and processing overhead.

### *4.5. Integrated Latency Reduction Strategies*

The combined use of these techniques results in a comprehensive framework for low-latency real-time AI summarization. By optimizing ingestion pipelines, leveraging

advanced protocols and hardware, scaling dynamically, and applying efficient serialization and compression methods, the system achieves sub-second response times even under high traffic conditions. These strategies ensure that users experience fast and reliable summarization services, meeting the stringent performance requirements of modern conversational AI applications.

**5. Achieving High Availability and Disaster Recovery**

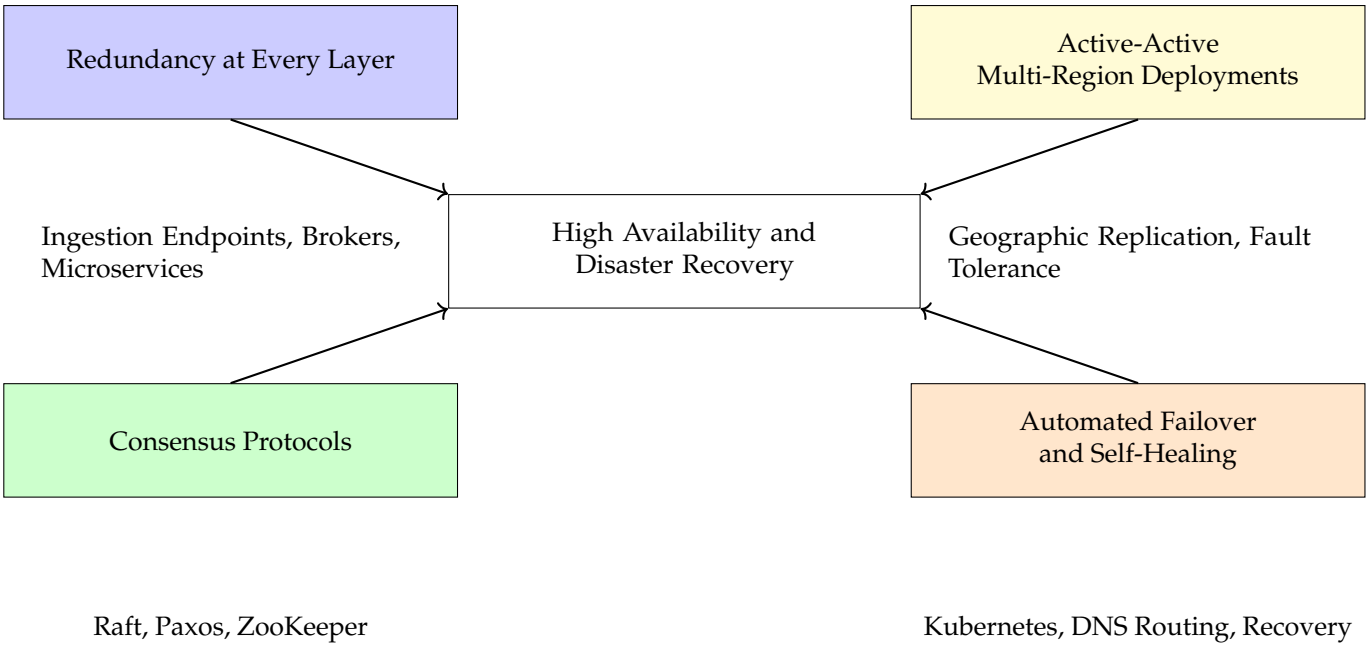


**Figure 6.** Achieving High Availability and Disaster Recovery: The system incorporates redundancy at all critical layers, active-active multi-region deployments for fault tolerance, consensus protocols for state synchronization, and automated failover mechanisms. These strategies ensure uninterrupted operation and rapid recovery from failures, minimizing downtime and preserving data integrity.

Ensuring high availability (HA) and robust disaster recovery (DR) mechanisms is essential for maintaining uninterrupted service in the face of component failures, network outages, or catastrophic events. This section outlines the strategies employed to achieve fault tolerance, minimize downtime, and ensure data integrity and service continuity. The architecture incorporates redundancy, distributed consensus protocols, and automated failover mechanisms to meet stringent uptime requirements.

*5.1. Redundancy at Every Layer*

At the core of high availability is the principle of redundancy, which involves deploying multiple instances of every critical system component. In the data ingestion layer, multiple geographically distributed endpoints collect and route data, ensuring that failure in one location does not disrupt data flow. Streaming platforms, such as Kafka, are configured with redundant brokers and replicated partitions. Each partition has multiple replicas stored across broker nodes to provide resilience against individual node failures. Similarly, summarization microservices are deployed as stateless, horizontally scalable instances, enabling any instance to take over the workload of a failed peer.

Geographic distribution of redundant components adds an additional layer of protection. For instance, ingestion endpoints and summarization services are deployed in different cloud regions or data centers. If one region experiences a failure, another can continue processing requests seamlessly. Load balancers and DNS configurations automatically detect the failure and reroute traffic to healthy instances, minimizing user-facing disruption.

Mirrored caches are also employed to replicate critical data, ensuring that cache-dependent workflows can continue operating even if one cache node becomes unavailable.

### *5.2. Active-Active Multi-Region Deployments*

Active-active multi-region deployments are a cornerstone of fault tolerance in distributed systems. In this configuration, multiple regions actively handle traffic simultaneously while maintaining synchronized states. This setup not only ensures fault tolerance but also improves latency for users by directing them to the closest operational region. Active-active replication involves complex challenges, particularly around maintaining consistency and preventing message duplication or loss.

Synchronization between active regions is achieved using advanced replication protocols that track offsets and ensure idempotent processing. For instance, Kafka's MirrorMaker or Confluent's multi-cluster replication tools synchronize data streams between regions while preserving message ordering. To handle failover, client applications are designed to automatically reconnect to the surviving region if their primary region becomes unavailable. This approach ensures minimal downtime and a seamless user experience.

A key consideration in active-active architectures is the trade-off between consistency and availability. By adopting a design that prioritizes eventual consistency and idempotent operations, the system avoids performance bottlenecks during inter-region synchronization while ensuring data integrity. Furthermore, this design mitigates the risk of split-brain scenarios, where multiple regions independently process the same data due to a loss of synchronization.

### *5.3. Consensus Protocols for State Synchronization*

Distributed consensus protocols, such as Raft and Paxos, play a vital role in maintaining a consistent view of the system state across replicas. These protocols enable coordinated decision-making even in the presence of network partitions or component failures. In streaming platforms like Kafka, the leader-follower replication model is a practical application of distributed consensus. When a partition leader fails, a new leader is elected through a quorum-based voting mechanism, ensuring continuity in data processing.

Consensus protocols are also used to manage cluster metadata and configuration changes in fault-tolerant key-value stores like ZooKeeper or etcd. These stores act as the backbone for many distributed systems, storing critical information such as partition assignments, leader elections, and service discovery data. By ensuring consistency across replicas, consensus protocols prevent conflicting or stale configurations that could disrupt service operation.

For summarization microservices, state synchronization is often simplified due to their stateless nature. However, in cases where session or cache data is shared, distributed caching platforms like Redis Cluster or Hazelcast use consistent hashing and quorum-based replication to synchronize data across nodes. This ensures that data remains accessible even during partial system failures.

### *5.4. Automated Failover and Self-Healing*

Automation is a critical aspect of high availability, enabling systems to detect failures and recover without human intervention. Automated failover mechanisms ensure that traffic is rerouted away from failed components to healthy ones. For instance, Kubernetes orchestrates failover by monitoring the health of pods through liveness and readiness probes. When a summarization engine instance or broker node fails, Kubernetes removes it from service and spins up a replacement instance. Similarly, load balancers, such as NGINX or HAProxy, dynamically adjust their routing rules to exclude failed nodes.

DNS-based failover mechanisms complement these strategies by redirecting traffic to healthy regions or endpoints during regional outages. Techniques such as weighted DNS or GeoDNS allow traffic to be routed based on availability and proximity, further improving resilience and performance.

Self-healing capabilities are embedded across the architecture to streamline recovery processes. For example, when a broker node crashes, Kafka automatically promotes a replica to leader status and resumes operations with minimal disruption. Similarly, distributed caches employ eviction and synchronization mechanisms to rebuild cache states after node failures. Orchestration platforms monitor resource usage and ensure that components are restarted or rescheduled on healthy nodes when hardware failures occur.

**Table 3.** Key High Availability Strategies and Their Benefits

| **Strategy** | **Description and Benefits** |
|---|---|
| Redundant Components | Multiple instances of ingestion nodes, brokers, and summarization microservices ensure that failure in one component does not disrupt overall operations. |
| Active-Active Deployments | Multi-region setups actively handle traffic in parallel, providing fault tolerance and improved latency for users. |
| Consensus Protocols | Distributed protocols like Raft and Paxos ensure consistency and reliable state synchronization across replicas. |
| Automated Failover | Orchestrators and DNS failover mechanisms automatically detect failures and reroute traffic to healthy instances, minimizing downtime. |
| Self-Healing Systems | Systems automatically restart failed nodes and rebuild states, reducing the need for manual intervention. |

### *5.5. Disaster Recovery Mechanisms*

In addition to high availability, disaster recovery mechanisms are essential for protecting against catastrophic failures, such as data center outages or large-scale network disruptions. The architecture employs periodic data backups, cross-region replication, and recovery drills to ensure that services can be restored quickly.

Data backups are taken at regular intervals and stored in geographically distributed locations to guard against data loss. Streaming platforms like Kafka maintain replicated logs, which can be replayed to rebuild system states during recovery. Summarization engines, which rely on pre-trained models, store model weights and checkpoints in object storage services with cross-region replication enabled.

Disaster recovery testing is a critical practice for ensuring readiness. Regular drills simulate scenarios such as region-wide outages, validating the system's ability to failover and recover within predefined recovery time objectives (RTOs). By continuously refining these processes, the architecture achieves both resilience and operational continuity.

**Table 4.** Disaster Recovery Mechanisms and Objectives

| **Mechanism** | **Description and Objective** |
|---|---|
| Cross-Region Backups | Data and system states are replicated across geographically distributed locations to ensure recovery in case of regional failures. |
| Replicated Streaming Logs | Kafka and similar platforms replay logs to rebuild states and recover lost messages during outages. |
| Disaster Recovery Drills | Simulated failures test the system's ability to meet recovery time objectives and maintain service continuity. |
| Model Checkpointing | AI models and summarization engine weights are stored in distributed object storage, ensuring they can be restored quickly during recovery. |

## 6. Performance Evaluation

In order to assess the effectiveness, robustness, and efficiency of the proposed resilient real-time data delivery architecture for AI summarization in conversational platforms, we conducted a comprehensive performance evaluation. Our primary goals were to measure end-to-end latency, throughput, availability under failure scenarios, and the speed and reliability of disaster recovery processes. To this end, we built a carefully controlled testbed that emulated a global deployment of the system, enabling us to investigate the behavior of each component under realistic conditions. Subjecting the pipeline to varying workloads, network perturbations, node failures, and regional outages, we obtained actionable results into system performance and areas for future optimization.

### *6.1. Testbed Description*

To validate the proposed architecture, we constructed a distributed testbed designed to mimic a global conversational AI platform serving geographically dispersed clients. The testbed spanned two cloud regions—US-East and EU-West—selected to represent realistic transatlantic latencies and network conditions. Each region hosted an identical stack of components, including ingestion endpoints, a Kafka-based streaming layer, caching systems, and containerized summarization services managed by Kubernetes. This dual-region setup allowed us to investigate both localized failure scenarios and larger-scale regional outages, providing insights into the system's active-active replication and failover mechanisms.

At the ingestion layer, we deployed multiple endpoint servers designed to receive conversational data from synthetic client simulators. These simulators generated messages at controlled rates, varying from moderate (10,000 messages per second) to peak (100,000 messages per second) workloads. The ingestion endpoints serialized incoming messages using Protocol Buffers and batched them in short, millisecond-scale micro-batches before sending them into the Kafka cluster. In each region, we operated a three-broker Kafka cluster configured for triple replication of topic partitions. This replication strategy ensured fault tolerance at the broker level and guaranteed data durability even if one or two brokers became unavailable.

Each Kafka cluster was supported by in-memory caching layers (using Redis) for frequently requested conversation segments and recently produced summaries. These caches reduced load on the summarization services and minimized re-processing times, ultimately contributing to lower end-to-end latency. The summarization layer itself consisted of containerized microservices running Transformer-based language models fine-tuned for dialogue summarization. These models were GPU-accelerated and horizontally scalable, allowing us to dynamically increase capacity when message rates spiked.

On top of these services, we implemented various optimizations and protocols aimed at minimizing latency and ensuring high availability. We leveraged gRPC over HTTP/2 for efficient message transport, reduced overhead with selective compression and zero-copy data transfer, and utilized Kubernetes Horizontal Pod Autoscalers for dynamic scaling of summarization services. Additionally, load balancers with integrated health checks rerouted traffic around failing components, while continuous monitoring and observability tools (e.g., Prometheus and Grafana) recorded performance metrics, latencies, and error rates.

To measure performance, we focused on the end-to-end latency, defined as the time from message injection at the ingestion endpoint until the summarization result was returned to the client simulator. We also examined throughput metrics—how steadily the system could process tens or hundreds of thousands of messages per second—under varying conditions. To assess availability, we simulated partial failures at different layers, including broker node crashes and summarization service interruptions. For disaster recovery evaluations, we introduced controlled regional outages, data corruption events, and observed how quickly and accurately the system could restore service and data consistency.

*6.2. Latency, Throughput, and Availability Benchmarks*

Our first series of experiments focused on evaluating system performance under nominal and peak loads, while introducing controlled component failures to measure availability. Under a nominal load of approximately 10,000 messages per second, the system consistently achieved a p95 (95th percentile) latency of under 80 ms. This included time for ingestion, Kafka write and read operations, data deserialization, summarization processing, and delivery of results. The low latency at this moderate load level was a direct result of careful architectural choices: using efficient serialization formats, leveraging micro-batching to amortize overheads, and employing zero-copy network transfers. The caching layer further contributed by enabling summarization engines to quickly access recent conversation context without re-fetching or re-processing large amounts of data.

As we gradually increased the traffic to a peak load of 100,000 messages per second, the system's latency naturally grew due to the increased pressure on network bandwidth, CPU/GPU resources, and I/O operations. Even so, the p95 latency remained around 120 ms, still comfortably within our target of keeping latency under 150 ms under peak conditions. The ability to scale the summarization services horizontally was critical here. When the system detected growing backlogs or prolonged processing times, Kubernetes automatically provisioned additional summarization pods. This scaling ensured that the pipeline maintained high throughput without saturating any single component.

We also investigated availability by simulating single-node broker failures. By forcibly terminating one of the Kafka brokers in the cluster, we observed the cluster's response time and effects on latency. The broker election protocols in Kafka triggered within 2–3 seconds, automatically choosing a new leader for the affected partitions. During this brief re-election window, we noted a slight spike in latency (on the order of tens of milliseconds), but crucially, no data loss occurred. The intact replicas of each partition ensured continuity, and once the new leader was elected, the system resumed normal operation.

We repeated a similar procedure for summarization services by forcibly crashing a subset of pods. Kubernetes promptly detected these failures through liveness probes and spun up new pods to maintain the desired replication level. The load balancer, observing failed health checks, seamlessly redirected requests to healthy pods. As a result, there was negligible downtime and only a minor latency hiccup as traffic was rerouted. Such resilience at the service layer illustrated the value of container orchestration and active load management, which, combined with reliable data streaming, kept the system running smoothly even under component-level failures.

Our benchmark results demonstrated that under nominal conditions, our architecture could reliably deliver low-latency responses at moderate to high traffic volumes. Under peak load, while latency did increase, it remained within acceptable bounds, thanks to adaptive scaling and careful optimization. Moreover, the system's high availability was confirmed by the absence of data loss and minimal downtime following single-node broker and summarization service failures.

*6.3. Disaster Recovery Experiments*

While single-node failures are common scenarios that any robust system should handle gracefully, larger-scale disasters pose more substantial challenges. To test the system's resilience under more catastrophic events, we conducted disaster recovery experiments involving regional outages and data corruption scenarios. These tests were designed to push the architecture's geo-redundancy, snapshot-based backups, and synchronization protocols to their limits, thereby validating the efficacy of our active-active replication approach and data durability strategies.

First, we simulated a regional outage by disabling the entire US-East region. This emulation involved shutting down all components—ingestion endpoints, Kafka brokers, caches, and summarization services—in that region. Under normal circumstances, such a large-scale failure could cause prolonged downtime and potentially significant data loss. However, our architecture was designed with active-active replication, meaning that both

US-East and EU-West regions were continuously receiving and processing data in parallel. The replicated Kafka topics ensured that EU-West had up-to-date copies of all conversation data and message offsets, allowing it to continue serving client requests without missing messages.

We observed that following the US-East shutdown, clients connected from various geographic locations automatically rerouted to the EU-West endpoints. This load redistribution caused only a slight increase in latency—approximately 15 ms on average—as the longer network paths for US-based users to EU-West servers contributed to the delay. Despite this slight increase, there was no downtime or data loss. The summarization services in EU-West continued producing real-time summaries, and the architecture's failover logic ensured uninterrupted operation. After we re-enabled the US-East region, the system's synchronization mechanisms quickly restored data consistency. Within a few minutes, offsets, caches, and cluster metadata were fully synchronized, and traffic load balanced between the two regions returned to its pre-failure distribution.

In addition to regional outages, we tested catastrophic data corruption scenarios. For instance, we simulated a situation where a subset of Kafka partitions in the EU-West region experienced logical corruption due to a faulty storage module. Without robust recovery mechanisms, such an event could lead to partial data loss or inconsistencies that degrade summarization quality. Our solution involved maintaining periodic snapshot-based backups of Kafka topics and associated metadata in geo-distributed object storage. Triggering a rollback from these snapshots restored the system's consistent state within approximately five minutes. This process included retrieving the backups, reinitializing Kafka topics to a known good offset, and allowing summarization services to replay recent data as needed.

Although the snapshot-based recovery introduced a longer downtime than simple node failures—indeed, five minutes represents a non-trivial interruption in a real-time setting—it ensured data integrity and continuity. Throughout this rollback, the system emitted clear status signals and error messages, allowing operators to track recovery progress. Once the data was restored, the summarization pipeline resumed its normal operations, confirming that our recovery strategy protected against even severe logical failures. While further optimization could reduce this recovery time, the key takeaway was that the architecture prevented permanent data loss and allowed for eventual consistency restoration after catastrophic events.

### *6.4. Discussion*

The suite of experiments and their results provided multiple valuable insights into how to build and operate a resilient, low-latency real-time summarization platform. First and foremost, the tests underscored the importance of careful orchestration at every layer. The combination of container orchestration tools like Kubernetes, streaming platforms like Kafka with robust replication and leader election mechanisms, and load balancing logic capable of rerouting requests dynamically created a system that could self-heal from component-level failures. These experiments showed that by layering redundancy and employing consensus protocols, we could isolate failures and prevent them from escalating into major outages.

Second, the results highlighted that low latency and high resiliency need not be mutually exclusive. At moderate loads, the pipeline maintained latencies of well under 100 ms, which is crucial for real-time user experiences. Even as the message rate soared to ten times the nominal load, adaptive scaling and efficient encoding formats kept latencies near 120 ms. This demonstrated that with the right architectural choices—such as micro-batching, zero-copy networking, and efficient serialization—maintaining millisecond-level response times at scale is achievable. These optimizations can coexist harmoniously with fault tolerance mechanisms, ensuring that adding resiliency does not inherently degrade performance.

Third, the experiments reinforced the value of elasticity. Workloads in conversational AI platforms are rarely static; user activity can spike unpredictably due to events like marketing campaigns, time-zone-driven usage peaks, or breaking news. The ability to spin up new summarization pods and scale horizontally in response to backlogs was instrumental in preserving performance under stress. Without elasticity, the system would either over-provision resources at all times—leading to cost inefficiencies—or accept degraded latency during peak load. Our tests showed that the adaptive strategies implemented struck a good balance, allowing the infrastructure to handle surges effectively without consistently running on costly overcapacity.

Furthermore, the disaster recovery experiments offered critical lessons on the design of multi-region, active-active deployments. The results indicated that geo-redundancy, if properly implemented, can ensure continuous operation even when an entire region is taken offline. This capability is invaluable when facing real-world disasters such as natural events, large-scale power outages, or substantial network partitions. Although complete regional failures are rare, preparing for them ensures higher availability and business continuity, which can significantly strengthen user trust and platform reputation.

The snapshot-based recovery from data corruption scenarios also taught us that while proactive replication and geo-distribution can mitigate many issues, maintaining offline, point-in-time backups remains essential. Logical errors, bugs in code, or operator mistakes can introduce corrupt data that replication alone cannot fix. Having a well-tested restoration process—from regularly scheduled snapshot creation to swift retrieval from object stores—proved vital. The five-minute restoration period, though not negligible, represented a substantial improvement over what could have been hours or days of downtime without such backups. Future efforts could focus on accelerating this recovery loop through incremental snapshots, partial replays, or more efficient metadata storage.

From an operational standpoint, the experiments illustrated that robust observability and monitoring are paramount. Detailed metrics allowed us to quickly pinpoint latency spikes, identify underutilized resources, and detect early signs of trouble in the pipeline. Proactive alerting enabled quick responses to failures, sometimes even automating remediation steps. Moreover, the ability to simulate failures—both partial (single node) and catastrophic (full region)—in a controlled testbed environment paved the way for continuous improvement. Regularly scheduled chaos engineering exercises could further strengthen the pipeline's resilience, as developers refine failover strategies, optimize caching layers, or tweak compression settings to handle exceptional conditions more gracefully.

The performance evaluation experiments confirmed that the proposed architecture can simultaneously deliver low latency, high availability, and robust disaster recovery capabilities. The combination of distributed streaming frameworks, container orchestration, adaptive scaling, zero-copy networking, schema-efficient serialization, geo-redundancy, and snapshot-based recovery mechanisms create a system capable of meeting the stringent demands of real-time conversational AI platforms. The lessons learned from these experiments will guide ongoing enhancements, such as optimizing backup frequency, reducing failover times, and refining auto-scaling policies. Ultimately, this ongoing process of measurement, experimentation, and continuous improvement ensures that the platform can confidently adapt to workloads, unforeseen failures, and new challenges in delivering real-time summarized insights at a global scale.

## 7. Conclusion

Real-time conversational AI platforms present unique challenges that lie at the intersection of data delivery, distributed systems, and AI model integration. Addressing these challenges requires a multidisciplinary approach that combines insights from streaming architectures, fault-tolerant system design, and advancements in natural language processing. This work contributes to the growing body of research by focusing on the infrastructural aspects of real-time summarization systems, complementing existing advances in AI and distributed computing.

A significant portion of the literature on real-time data delivery systems has explored platforms such as Apache Kafka, Pulsar, and Flink, which are designed to handle high-throughput, low-latency streaming workloads. These systems emphasize techniques such as partitioning, replication, and back-pressure mechanisms to maintain scalability and reliability in distributed environments. Partitioning strategies improve throughput by enabling parallel processing, while replication ensures data availability even in the event of node failures. Back-pressure mechanisms, critical in preventing data overflow and cascading failures, allow upstream components to adjust their rates based on downstream capacity.

Edge computing has also gained attention as a means to reduce latency and bandwidth requirements. Recent work has focused on deploying geographically distributed pipelines for edge data processing. However, much of this research centers around sensor data or IoT applications, with less focus on the nuanced requirements of conversational workloads, such as maintaining strict message ordering and providing consistent, context-aware data streams. This paper builds upon these foundations, adapting these principles to address the unique challenges posed by real-time summarization tasks.

On the AI side, the rise of transformer-based architectures, such as BERT and GPT, has revolutionized summarization tasks. These models achieve state-of-the-art performance through techniques such as pre-training on vast corpora and fine-tuning on domain-specific datasets. Research in this area primarily focuses on optimizing model architectures, improving data preprocessing pipelines, and leveraging transfer learning to adapt generic models to specific summarization tasks.

Despite these advancements, the infrastructural requirements of delivering real-time data to these models remain underexplored. For these models to perform optimally in real-time scenarios, they require a steady stream of high-quality, low-latency data. This work complements ongoing AI research by addressing the systemic challenges of ensuring that summarization models operate under ideal conditions. By designing resilient and efficient data pipelines, we ensure that these models can consistently generate accurate and timely summaries, even in dynamic and fault-prone environments.

The fields of fault tolerance and disaster recovery have been extensively studied in distributed systems. Traditional approaches such as quorum-based replication, write-ahead logging, and eventual consistency have laid the groundwork for modern high-availability frameworks. More recently, systems like Spanner, CockroachDB, and YugabyteDB have introduced globally consistent database solutions, employing techniques such as TrueTime synchronization and geo-distributed transactions to achieve strong consistency across regions.

While these frameworks address many challenges associated with stateful, transactional workloads, our work focuses on streaming and real-time data pipelines, which require a different set of optimizations. Unlike transactional systems, real-time pipelines must prioritize low-latency processing and throughput while maintaining consistency and fault tolerance. By adapting concepts from database resilience, such as active-active geo-replication and quorum-based leader election, we design a system tailored to the specific demands of summarization workloads. This ensures that the system remains consistent, durable, and highly available, even during large-scale failures.

This paper has presented an architecture and methodologies for achieving resilient, real-time data delivery tailored to AI-driven summarization in conversational platforms. We identified key challenges—low latency, high availability, and disaster recovery—and proposed solutions that integrate streaming platforms, distributed consensus, active-active replication, caching, load balancing, and container orchestration. Experimental results show that our approach can reliably meet stringent latency targets and maintain availability even under severe failure scenarios.

Future research directions include exploring adaptive routing algorithms that dynamically choose the optimal region or pipeline stage for processing incoming data, integrating more advanced anomaly detection techniques for proactive failure mitigation, and inves-

tigating the interplay of cost optimization strategies in multi-cloud environments. As conversational systems become increasingly integral to modern communication, ensuring the resilience and efficiency of their underlying data delivery infrastructure remains a critical area of inquiry.

## References


1. Yang, M.; Li, C.; Sun, F.; Zhao, Z.; Shen, Y.; Wu, C. Be relevant, non-redundant, and timely: Deep reinforcement learning for real-time event summarization. In Proceedings of the Proceedings of the AAAI Conference on Artificial Intelligence, 2020, Vol. 34, pp. 9410–9417.
2. Babu, N.T.; Stewart, C. Energy, latency and staleness tradeoffs in ai-driven iot. In Proceedings of the Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, 2019, pp. 425–430.
3. Deng, C.; Fang, X.; Wang, X.; Law, K. Software orchestrated and hardware accelerated artificial intelligence: toward low latency edge computing. *IEEE Wireless Communications* **2022**, *29*, 110–117.
4. Wang, Y.; Dong, Y.; Guo, S.; Yang, Y.; Liao, X. Latency-aware adaptive video summarization for mobile edge clouds. *IEEE Transactions on Multimedia* **2019**, *22*, 1193–1207.
5. Tang, Y.; Puduppully, R.; Liu, Z.; Chen, N. In-context learning of large language models for controlled dialogue summarization: A holistic benchmark and empirical analysis. In Proceedings of the Proceedings of the 4th New Frontiers in Summarization Workshop, 2023, pp. 56–67.
6. Gupta, D.; Bhatia, M.; Kumar, A. Resolving data overload and latency issues in multivariate time-series IoMT data for mental health monitoring. *IEEE Sensors Journal* **2021**, *21*, 25421–25428.
7. Jiang, X.; Shokri-Ghadikolaei, H.; Fodor, G.; Modiano, E.; Pang, Z.; Zorzi, M.; Fischione, C. Low-latency networking: Where latency lurks and how to tame it. *Proceedings of the IEEE* **2018**, *107*, 280–306.
8. Santos, J.; Wauters, T.; Volckaert, B.; De Turck, F. Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions. *IEEE Communications Surveys & Tutorials* **2021**, *23*, 2557–2589.
9. Jiang, Z.; Fu, S.; Zhou, S.; Niu, Z.; Zhang, S.; Xu, S. AI-assisted low information latency wireless networking. *IEEE Wireless Communications* **2020**, *27*, 108–115.
10. Liu, D.; Sun, F.; Wang, W.; Dev, K. Distributed computation offloading with low latency for artificial intelligence in vehicular networking. *IEEE Communications Standards Magazine* **2023**, *7*, 74–80.
11. Mutalemwa, L.C.; Shin, S. A classification of the enabling techniques for low latency and reliable communications in 5G and beyond: AI-enabled edge caching. *IEEE Access* **2020**, *8*, 205502–205533.
12. Ma, J.; Li, T.; Zhang, Y. AI-based abstractive text summarization towards AIoT and edge computing. *Internet Technology Letters* **2023**, *6*, e354.
13. Lim, J. Latency-aware task scheduling for IoT applications based on artificial intelligence with partitioning in small-scale fog computing environments. *Sensors* **2022**, *22*, 7326.
14. Richardson, C.; Zhang, Y.; Gillespie, K.; Kar, S.; Singh, A.; Raeesy, Z.; Khan, O.Z.; Sethy, A. Integrating summarization and retrieval for enhanced personalization via large language models. *arXiv preprint arXiv:2310.20081* **2023**.