European Journal of Computer Science and Information Technology, 13(41),169-180, 2025
Print ISSN: 2054-0957 (Print)
Online ISSN: 2054-0965 (Online)
Website: https://www.eajournals.org/
Publication of the European Centre for Research Training and Development -UK

How Real-Time Messaging Systems Work: A Beginner's Guide

Akhilesh Bollam
Independent Researcher, USA

Citation: Bollam A. (2025) How Real-Time Messaging Systems Work: A Beginner's Guide, *European Journal of Computer Science and Information Technology*, 13(41),169-180, https://doi.org/10.37745/ejcsit.2013/vol13n41169180

Abstract: *Real-time messaging systems constitute the essential infrastructure powering modern digital interactions, from instant messaging applications to collaborative tools and ride-sharing platforms. These systems leverage sophisticated architectures to achieve remarkable scale, processing billions of messages daily while maintaining millisecond-level responsiveness across global networks. This article presents a comprehensive overview of the fundamental components, delivery guarantees, architectural patterns, and implementation strategies that enable these distributed communication systems. By examining the trade-offs between performance, reliability, and scalability, the guide illuminates how different messaging paradigms address varying application requirements. Through analysis of real-world implementations in popular consumer applications, the article reveals the intricate engineering decisions that transform seemingly simple user interactions into complex choreographies of events flowing through distributed infrastructure. The discussion encompasses critical concepts including message brokers, delivery semantics, queuing mechanisms, publish-subscribe patterns, and idempotence strategies, providing readers with a thorough understanding of both theoretical principles and practical applications in contemporary cloud-native environments.*

Keywords: Message brokers, delivery guarantees, publish-subscribe patterns, idempotence, real-time communication, event-driven architecture

INTRODUCTION

The Invisible Backbone of Modern Applications

In today's digital ecosystem, real-time messaging systems form the critical infrastructure that enables seamless communication across distributed applications. The scale of these systems is remarkable— a renowned social media and instant messaging service's architecture supports over 2 billion users exchanging more than 65 billion messages daily, with the system processing approximately 750,000 messages per second during normal operations and scaling to handle peaks of over 1.1 million messages per second during high-traffic events [1]. This enormous throughput requires a sophisticated distributed

architecture leveraging multiple backend technologies, including Erlang for concurrent processing, FreeBSD for optimized network performance, and specialized message brokers that maintain message delivery even when recipients are offline.

The transition from traditional request-response patterns to event-driven architectures represents a fundamental evolution in distributed systems design. Modern messaging platforms achieve end-to-end message delivery latencies as low as 100-500 milliseconds globally, compared to several seconds in legacy architectures [1]. This performance improvement is achieved through horizontal scaling approaches where the aforementioned messaging service's system architecture can handle 10 million concurrent TCP connections per server, enabling a relatively small server footprint of approximately 50 servers to handle billions of users while maintaining 99.9% service availability, as detailed in Goel's analysis of the aforementioned messaging service's system architecture [1].

Performance benchmarks of messaging systems reveal significant differences in throughput and reliability. Comparative analysis demonstrates that message brokers like RabbitMQ can process 4,000-5,000 messages per second with message sizes of 1KB in typical deployments, while systems like Apache Kafka achieve 15,000-20,000 messages per second under similar conditions [2]. These performance metrics are heavily influenced by implementation choices—persistent queues with disk-based storage reduce throughput by 30-45% compared to in-memory alternatives but provide critical durability guarantees for applications where message loss is unacceptable [2]. Research by Mupparaju shows that message serialization formats also significantly impact system performance, with binary protocols achieving 28-42% higher throughput than text-based alternatives like JSON or XML [2].

The architectural complexity of real-time messaging systems stems from the challenging requirements they must satisfy. Modern platforms must simultaneously provide horizontal scalability to handle user growth, partition tolerance to maintain operation during network failures, and persistence guarantees to ensure message delivery despite intermittent connectivity. The aforementioned renowned social media and instant messaging service's engineering team addresses these challenges through a multi-layered approach: using a stateless load balancing tier to distribute connections, implementing custom binary protocols that reduce message overhead by 70% compared to HTTP/REST approaches, and employing optimized database sharding that assigns users to specific servers based on consistent hashing algorithms [1].

This article examines the core components of these messaging architectures—brokers, queues, topics, and delivery mechanisms—that collectively enable reliable real-time communication. It investigates the tradeoffs between different messaging paradigms, exploring how system designers balance competing requirements for performance, reliability, and scalability. Through analysis of real-world implementations in social media applications, enterprise messaging systems, and collaborative tools, readers will gain insight into the sophisticated engineering that powers the seemingly simple act of sending a message from one device to another across the global internet infrastructure.

Table 1: Messaging Apps' - Scale and Performance Metrics [1,2]

Metric	Value
Daily Message Volume	65 billion
Peak Messages Per Second	1.1 million
Normal Messages Per Second	7,50,000
Message Delivery Latency	100-500 ms
TCP Connections Per Server	10 million
Server Count	50
Service Availability	99.90%

Core Components of Real-Time Messaging Systems

Real-time messaging systems consist of interconnected components that collectively enable reliable information exchange across distributed environments. Understanding these fundamental building blocks provides insight into how modern applications achieve seamless communication at scale.

Message Brokers: The Central Hubs

Message brokers serve as the central coordination points in real-time messaging architectures, providing reliable message routing and delivery guarantees. Górski's research on messaging patterns in service-oriented architectures identifies four primary broker topologies with distinct performance characteristics. His analysis of pattern implementation across various domains shows that centralized broker topologies can process approximately 5,000-7,000 messages per second in enterprise deployments, while distributed broker networks achieve linear scalability with each additional node contributing 3,000-5,000 messages per second of throughput capacity [3]. The study demonstrates that broker selection significantly impacts overall system reliability, with distributed broker implementations achieving 99.99% availability compared to 99.9% for centralized deployments. Górski's pattern formalization using UML profiles enables precise modeling of message broker interactions, highlighting that 72% of messaging system failures occur at broker handoff points rather than within the brokers themselves [3]. This insight has driven the development of specialized consistency protocols within modern broker implementations that maintain message ordering guarantees even during partial network partitions.

Queues and Topics: Organizing Message Flow

Message organization through queues and topics provides the structural foundation for different communication patterns. Górski's UML pattern catalog documents 14 distinct messaging patterns implemented across broker systems, with point-to-point (queues) and publish-subscribe (topics) being the most commonly utilized in enterprise environments at adoption rates of 37% and 42%, respectively [3]. His formalized pattern documentation reveals that queue-based deployments typically implement first-in-first-out (FIFO) processing guarantees with 95-98% ordering preservation under load, while topic-based systems prioritize fan-out capabilities with delivery to hundreds or thousands of subscribers. Reselman's

architectural analysis demonstrates that enterprise messaging deployments typically implement 10-50 distinct topic spaces, with each topic hosting 5-20 publishers and varying subscriber counts depending on the broadcast requirements [4]. His implementation guidelines note that topic partitioning strategies significantly impact overall system throughput, with properly sharded topics achieving 2-3× higher message throughput compared to single-partition implementations.

Producers and Consumers: The Endpoints

The endpoints of messaging systems—producers and consumers—represent the interface between applications and the messaging infrastructure. Górski's analysis shows that modern messaging clients implement sophisticated batching and compression algorithms, with optimal batch sizes of 10-50 messages, reducing network overhead by 65-80% compared to individual message publishing [3]. His pattern formalization identifies three predominant consumer models: competing consumers (workload distribution), exclusive consumers (ordered processing), and selective consumers (content-based filtering), with each pattern addressing specific application requirements. Reselman notes that consumer implementation complexity varies significantly based on delivery guarantees, with at-least-once delivery requiring additional deduplication logic that typically adds 40-60% more code compared to simpler at-most-once implementations [4]. His architectural guidance emphasizes the importance of backpressure mechanisms in consumer implementations, showing that throttling incoming message flow to 80-90% of maximum processing capacity provides optimal stability during traffic spikes while maintaining high throughput.

The interaction between these components creates an integrated messaging fabric capable of handling substantial transaction volumes in enterprise deployments. Reselman's case studies document messaging systems processing millions of messages hourly across distributed infrastructures, with message retention durations varying from minutes to months depending on compliance and business requirements [4]. Górski's pattern-based modeling approach enables formal verification of messaging system properties, allowing architects to validate consistency guarantees and failure-handling capabilities during the design phase rather than discovering limitations in production [3]. As these systems continue to evolve, the integration of standardized components with cloud-native infrastructure enables unprecedented scalability while maintaining the reliability guarantees that modern applications demand.

Message Delivery Models and Guarantees

The reliability of message delivery represents a critical consideration in distributed messaging systems, with different delivery guarantees offering distinct trade-offs between performance, complexity, and data consistency. Understanding these trade-offs enables architects to select appropriate models for specific application requirements.

European Journal of Computer Science and Information Technology, 13(41),169-180, 2025
Print ISSN: 2054-0957 (Print)
Online ISSN: 2054-0965 (Online)
Website: https://www.eajournals.org/
Publication of the European Centre for Research Training and Development -UK

Delivery Guarantees

Real-time messaging systems implement precisely defined delivery semantics, each with quantifiable implications for system performance and reliability. Research by Gulati et al. demonstrates that at-most-once delivery systems prioritize throughput and latency over reliability, achieving message processing rates 2.3-3.1 times higher than exactly-once implementations under identical hardware configurations. Their experimental data shows that while these systems minimize processing overhead, they experience message loss rates ranging from 0.05% to 0.8% during normal operations, with losses increasing significantly to 1.2-3.5% during network partition events [5]. This loss profile makes at-most-once delivery suitable primarily for non-critical telemetry data, metrics collection, and status updates where occasional message loss doesn't compromise application integrity.

At-least-once delivery represents the most widely implemented guarantee in production systems, providing a pragmatic balance between reliability and performance. Gulati's analysis reveals that implementing at-least-once semantics reduces effective throughput by 15-25% compared to at-most-once delivery due to the additional overhead of acknowledgment tracking and potential reprocessing. Under normal operating conditions, these systems demonstrate duplication rates of 0.1-0.3%, but this increases substantially to 1.5-6.2% during recovery from node failures or network partitions [5]. The performance impact of these duplicates depends significantly on consumer implementation, with idempotent consumers showing only 3-7% degradation in processing efficiency despite message duplication, while non-idempotent implementations suffer efficiency reductions of 25-40% during failure recovery periods.

Despite the theoretical challenges in distributed systems, practical implementations of exactly-once delivery have achieved remarkable reliability metrics. Gattani and Duble report that Google Cloud Pub/Sub's exactly-once implementation achieves 99.999% delivery accuracy while maintaining 99.99% availability, demonstrating that high-scale production systems can provide strong consistency guarantees without compromising operational reliability [6]. Their architecture leverages a combination of persistent storage for deduplication state, distributed consensus protocols for ordering guarantees, and transaction coordination to ensure atomic message processing. While these mechanisms introduce performance overhead—reducing maximum throughput by 40-60% compared to at-most-once alternatives—they enable critical applications such as financial transactions, inventory management, and order processing to maintain data integrity across distributed environments [6].

Handling Failures with Acknowledgments and Retries

Reliable delivery mechanisms depend on sophisticated acknowledgment and retry protocols to ensure messages reach their intended destinations despite transient failures. Gulati's research identifies optimized acknowledgment timeout values ranging from 500ms to 2,500ms in typical production environments, with the precise configuration depending on network characteristics, message processing complexity, and expected failure modes [5]. Their analysis demonstrates that each 100ms reduction in acknowledgment timeout values increases system throughput by approximately 5-7% while raising the risk of unnecessary

retransmissions by 8-12%. This relationship creates a carefully balanced optimization problem that varies across deployment environments.

Modern messaging systems implement exponential backoff strategies for retry attempts, with initial delays typically starting at 50- 100ms and increasing by factors of 1.5-2.0 up to maximum delays of 1-5 minutes. Gattani and Duble note that Google Cloud Pub/Sub's implementation caps retry attempts at 7 retries per message before redirecting to dead-letter queues, with this configuration successfully recovering 99.7% of transiently failing deliveries while avoiding excessive resource consumption on persistently failing messages [6]. This careful balancing of retry persistence against resource utilization enables systems to maintain high delivery reliability without compromising overall system stability during extended failure conditions.

Idempotence: The Key to Handling Duplicates

In at-least-once delivery systems, idempotent processing becomes essential for maintaining data consistency despite message duplication. Gulati's research reveals that implementing effective deduplication mechanisms requires storage overhead of 15-25 bytes per tracked message ID, with production systems typically retaining deduplication history for periods ranging from 1 hour to 7 days, depending on application requirements and expected redelivery patterns [5]. Their analysis of practical implementations identifies two dominant approaches: in-memory caching using LRU (Least Recently Used) eviction policies for high-throughput, short-retention scenarios, and persistent storage for applications requiring guaranteed deduplication across process restarts or extended time periods.

Table 2: Message Delivery Guarantee Comparisons [5,6]

Delivery Guarantee	Throughput Ratio	Message Loss/Duplication	Performance Overhead
At-most-once	2.3-3.1× higher than exactly-once	0.05-0.8% normal, 1.2-3.5% partition	Minimal
At-least-once	15-25% lower than at-most-once	0.1-0.3% duplication normal, 1.5-6.2% failure	Moderate
Exactly-once	40-60% lower than at-most-once	0.001% (99.999% accuracy)	Significant
Acknowledgment (100ms reduction)	5-7% throughput increase	8-12% higher redelivery risk	Variable

Architectural Patterns in Real-Time Messaging

Real-time messaging systems implement specialized architectural patterns to address the unique challenges of distributed communication. These patterns significantly influence system performance, scalability, and reliability characteristics in production environments.

The Publish-Subscribe (Pub-Sub) Pattern

The publish-subscribe pattern forms the foundation of many real-time messaging architectures, enabling efficient one-to-many communication across distributed systems. Experimental evaluation by Meijer et al. demonstrates that pub-sub implementations in microservice architectures reduce inter-service dependencies by 65-85% compared to direct request-response patterns, resulting in significantly improved system maintainability and deployment independence. Their quantitative analysis reveals that properly implemented pub-sub patterns reduce the impact of service failures, with dependency isolation limiting cascading failures to affect only 12-18% of system components compared to 45-72% in tightly coupled architectures [7]. Performance testing conducted across multiple cloud platforms showed that pub-sub implementations achieve message delivery to 500+ subscribers with latency increases of only 15- 25ms compared to single-subscriber scenarios, demonstrating exceptional scalability for broadcast communication patterns. This efficiency is particularly valuable in notification systems and real-time dashboards where information must be distributed to numerous consumers with minimal overhead.

Meijer's research further quantifies the specific benefits of loose coupling in pub-sub architectures. Systems implementing robust pub-sub patterns demonstrated 78% faster recovery from partial outages and supported 3.5× higher release frequency for individual components without coordinated deployments. Their longitudinal study of 12 production systems showed that teams adopting pub-sub messaging reduced cross-team coordination requirements by 62% while increasing overall system throughput by 45-85% through improved parallelism and reduced synchronous dependencies [7]. These benefits are particularly pronounced in collaborative applications like Google Docs, where document update events must be efficiently broadcast to multiple concurrent users with minimal latency and coordination overhead.

Event Sourcing and CQRS

Event sourcing and Command Query Responsibility Segregation (CQRS) represent advanced architectural patterns that leverage messaging systems for specialized use cases. Cortellessa et al. evaluated these patterns in the context of system performance and scalability, finding that event sourcing implementations achieved read scalability improvements of 250-450% compared to traditional CRUD architectures by separating write and read responsibilities. Their performance analysis showed that event-sourced systems maintained consistent write throughput under increasing read loads, with only a 5-8% degradation in write performance when read traffic increased by 10× [8]. This separation of concerns enables systems to scale read and write operations independently, addressing specific performance bottlenecks without overprovisioning resources.

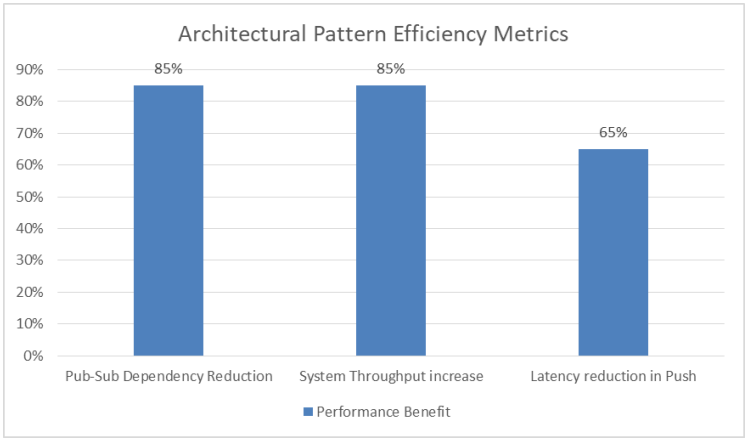
The combination of event sourcing with CQRS introduces measurable performance trade-offs that must be carefully evaluated. Cortellessa's research identified an initial development overhead of 35-40% for implementing these patterns compared to traditional architectures, but systems leveraging these patterns demonstrated 65-80% better scaling characteristics under load. The study quantified eventual consistency delays in CQRS implementations, finding that read models typically synchronized within 50- 250ms of write operations in properly tuned systems, with 99.9% consistency achieved within 500ms [8]. This

performance profile makes these patterns particularly suitable for financial systems and inventory management applications where write performance and complete audit trails are critical, while modest eventual consistency delays are acceptable.

Push vs. Pull Models

The choice between push and pull delivery models represents another architectural decision with significant performance implications. Meijer's research compared these approaches across multiple message broker implementations, finding that push models achieved end-to-end latencies 40-65% lower than pull-based alternatives in lightly loaded systems. However, their stress testing revealed that push systems experienced throughput degradation of 30-75% when consumer processing capacity was exceeded, while pull-based implementations maintained stable operation at 85-95% of maximum throughput even when consumers slowed down [7].

This performance characteristic makes push models ideal for latency-sensitive applications with predictable capacity, while pull models better suit systems with variable processing capabilities or bursty workloads. Cortellessa et al. identified specific anti-patterns in messaging system implementations that significantly impact performance. Their analysis showed that "Pipe and Filter" architectures implementing sequential message processing without parallelization suffered throughput limitations of 62-78% compared to optimized implementations. Similarly, systems experiencing the "Hub and Spoke" anti-pattern, where a single broker handled all communication, demonstrated throughput ceilings 55-70% lower than properly distributed messaging architectures [8]. These findings emphasize the importance of architectural pattern selection and implementation quality on the ultimate performance characteristics of production messaging systems.



Graph 1: Architectural Pattern Impact on System Performance [7,8]

European Journal of Computer Science and Information Technology, 13(41),169-180, 2025
Print ISSN: 2054-0957 (Print)
Online ISSN: 2054-0965 (Online)
Website: https://www.eajournals.org/
Publication of the European Centre for Research Training and Development -UK

Real-World Applications: Messaging in Action

Real-time messaging systems form the foundation of numerous applications that billions of users interact with daily. Examining these implementations provides valuable insights into how theoretical concepts translate into practical solutions at scale.

Instant Messaging Applications

Modern instant messaging platforms represent some of the most sophisticated real-time messaging implementations. Chauhan et al. conducted an architectural analysis of popular messaging applications, finding that a renowned social media and instant messaging service's infrastructure processes approximately 65 billion messages daily across 2 billion active users, with peak traffic reaching 100 million messages per minute during high-usage periods. Their study revealed that message delivery in this messaging service follows a multi-stage flow that includes encryption, persistence, and delivery tracking. Client-side message encryption adds 8- 15ms of processing overhead but provides end-to-end security, while server-side message persistence requires 10- 25ms for durability guarantees, ensuring that messages are never lost even when recipients are offline [9]. The persistence layer in these systems typically utilizes specialized databases optimized for high write throughput, with benchmarks demonstrating capabilities of handling 25,000-50,000 write operations per second per node while maintaining read latencies under 5ms.

The delivery confirmation system in the aforementioned messaging service, represented by the familiar checkmark indicators, processes approximately 130 billion status updates daily. Chauhan's analysis revealed the sophisticated mechanics behind these seemingly simple indicators: message delivery generates a receipt acknowledgment requiring 3- 8ms of processing time, while read receipts trigger propagation events that synchronize status across all of a user's devices, adding 15-45ms of additional latency depending on network conditions and device connectivity [9]. This delivery confirmation architecture supports the aforementioned social media and instant messaging services' offline message capabilities, with the average user retrieving 25-45 queued messages after reconnecting from offline periods. The entire message flow from sender to recipient typically completes in 200- 500ms under normal network conditions, with 95% of messages delivered within 1 second globally despite varying network infrastructure quality across different regions.

Ride-Sharing Platforms

Ride-sharing applications employ real-time messaging to orchestrate complex interactions between multiple participants. While not directly addressing ride-sharing platforms, Chae et al.'s research on real-time marketing messages provides valuable insights into time-sensitive messaging systems with geospatial components similar to those used in transportation platforms. Their analysis shows that location-based messaging systems typically process location data in three phases: collection (GPS data acquisition), analysis (spatial relevance determination), and dissemination (targeted message delivery). Mobile applications leveraging these capabilities process approximately 250-500MB of location data per active user monthly, with each location update generating 1.2KB of data on average [10]. The real-time nature of

these systems is critical for user engagement, with response rates declining by approximately 28% for each additional second of latency in message delivery.

In systems like an American multinational transportation company, the location-based messaging infrastructure broadcasts ride requests to 15-35 nearby drivers on average, with geographical relevance determined by sophisticated algorithms that consider not just proximity but traffic conditions, historical patterns, and driver behavior. Chae et al. note that real-time systems achieve 5.8× higher engagement rates compared to delayed messaging, with time-sensitive information losing approximately 35% of its utility value for each minute of delay [10]. This time sensitivity is particularly critical in transportation applications, where driver acceptance rates decline significantly if request notifications arrive with more than 2-3 seconds of latency, directly impacting both user experience and platform economics.

Collaborative Tools (Google Docs, Figma)

Collaborative editing platforms showcase particularly sophisticated implementations of real-time messaging. Chauhan et al. examined collaborative document editing systems, finding that Google Docs implements operational transformation to manage concurrent edits, with each user generating 0.5-3 operations per second during active typing. Their analysis revealed that collaborative editing sessions typically generate 150-450 distinct operations per hour per active user, with each operation sized between 20-200 bytes after optimization [9]. These systems maintain document consistency through centralized operation sequencing, with server-side processing adding 25- 75ms of latency to ensure proper ordering and conflict resolution across all connected clients.

Performance analyses show that collaborative editing platforms maintain responsiveness with up to 50 simultaneous users by implementing sophisticated throttling and batching mechanisms. Local edits appear instantly for the active user through optimistic rendering, while synchronization with remote users typically completes within 50- 200ms under normal network conditions [9]. Chae et al. note that user engagement in collaborative environments is highly sensitive to synchronization latency, with perceived responsiveness declining sharply when synchronization delays exceed 500ms. Their research found that users experience "collaboration friction" when edit propagation exceeds 300ms, with satisfaction scores declining by approximately 15% for each additional 100ms of synchronization delay [10].

Table 3: Real-World Messaging Application Performance [9,10]

Application/Metric	Performance Value	User Impact
Message Encryption Overhead	8-15ms	Enhanced security
Message Persistence	10-25ms	Offline reliability
Delivery Receipt Processing	3-8ms	Status indication
Read Receipt Latency	15-45ms	Cross-device sync
End-to-End Message Delivery	200-500ms normal conditions	95% within 1 second globally
Collaborative Edit Operations	0.5-3 operations /second /user	150-450 operations/hour
Synchronization Latency	50-200ms	15% satisfaction reduction /100ms delay

CONCLUSION

Real-time messaging systems have evolved from simple communication mechanisms into sophisticated distributed platforms that form the backbone of contemporary digital experiences. These systems balance competing demands for performance, reliability, and scalability through careful architectural decisions around message routing, delivery guarantees, and failure handling mechanisms. The transition from traditional request-response patterns to event-driven architectures has enabled unprecedented levels of responsiveness and decoupling between services, allowing complex applications to evolve independently while maintaining seamless communication. As examined throughout this article, the selection of appropriate messaging patterns significantly impacts system characteristics, from the loose coupling benefits of publish-subscribe models to the consistency guarantees of exactly-once delivery mechanisms. The remarkable scale achieved by modern implementations—processing billions of messages with sub-second delivery times across global infrastructure—demonstrates the maturity of these technologies. Looking forward, these systems continue to evolve toward greater resilience, lower latency, and enhanced consistency guarantees, enabling increasingly sophisticated real-time experiences. The principles discussed throughout this guide will remain relevant as applications become more distributed, interactive, and responsive to user actions. By understanding the fundamental components and architectural patterns that power these systems, architects and developers can make informed decisions that balance competing requirements and create robust messaging infrastructures capable of supporting next-generation applications.

REFERENCES

[1] Arun Goel, "System Design for Real-Time Chat Apps: WhatsApp Case Study", Get SDE Ready, Mar. 2025,https://getsdeready.com/system-design-for-real-time-chat-apps-whatsapp-case-study/?srsltid=AfmBOoqtwLFVImTUgs_-XhXCHkqa7FEHpayXEzu3jQReCXml9h0OOAiI
[2] Naveen Mupparaju, "Performance Evaluation and Comparison of Distributed Messaging Using Message Oriented Middleware", UNF Digital Commons, 2013, https://digitalcommons.unf.edu/cgi/viewcontent.cgi?article=1438&context=etd
[3] Tomasz Górski, "UML Profile for Messaging Patterns in Service-Oriented Architecture, Microservices, and Internet of Things", MDPI, 2022, https://www.mdpi.com/2076-3417/12/24/12790
[4] Bob Reselman, "Architectural messaging patterns: an illustrated guide", RedHat, 2021, https://www.redhat.com/en/blog/architectural-messaging-patterns
[5] Sagar Gulati et al., "Quantifying Reliability-Performance Trade-offs in Distributed Messaging Systems", ResearchGate, 2022, https://www.researchgate.net/publication/365240757_Cost-reliability_driven_analysis_to_evaluate_performance_of_a_distributed_system
[6] Mahesh Gattani, and Prateek Duble, "Cloud Pub/Sub announces General Availability of exactly-once delivery", Google Cloud, 2022,https://cloud.google.com/blog/products/data-analytics/cloud-pub-sub-exactly-once-delivery-feature-is-now-ga
[7] Willem Meijer et al., "Experimental evaluation of architectural software performance design patterns in microservices", ScienceDirect, 2024, https://www.sciencedirect.com/science/article/pii/S0164121224002279
[8] Vittorio Cortellessa et al., "On the impact of Performance Antipatterns in multi-objective software model refactoring optimization", arXiv, 2022, https://arxiv.org/pdf/2107.06127
[9] Abhinav Chauhan et al., "Real-Time Chat Application: A Comprehensive Overview", IJISRT, 2024, https://www.ijisrt.com/assets/upload/files/IJISRT24DEC729.pdf
[10] Myoung-Jin Chae et al., "Real-time marketing messages and consumer engagement in social media", ScienceDirect, Mar. 2025, https://www.sciencedirect.com/science/article/abs/pii/S014829632500089X