Технічні науки

УДК 004

Perih Anastasiia

Full Stack Software Engineer at Northspyre

Jersey City, NJ, US

ARCHITECTURAL SOLUTIONS FOR IMPLEMENTING REAL-TIME APPLICATIONS IN THE DIGITAL ENVIRONMENT

Summary. *This article explores architectural solutions and technological approaches for implementing real-time applications in the modern digital environment. The relevance of the topic is driven by the rapid growth of streaming data volumes and the need for their immediate processing, which requires a rethinking of traditional approaches and the application of new architectural models. The novelty of the research lies in the comprehensive analysis and systematization of various architectural solutions, such as microservices architecture, event-driven systems (EDA), reactive systems, edge computing, and the use of streaming platforms (Kafka, RabbitMQ). The paper describes the key requirements for real-time systems, including minimizing latency, fault tolerance, elasticity, and high throughput. The study examines widely used English-language sources from open access, including reviews, technical reports, and case studies from companies (Netflix, Amazon, banking systems, and trading platforms). Special attention is given to analyzing the advantages and disadvantages of each architectural approach, as well as practical recommendations for their implementation. The conclusion summarizes which combinations of architectural solutions are most effective for specific types of tasks. This article will be useful for architects, engineers, and specialists working with high-load real-time systems.*

Key words: *real-time applications, microservices architecture, event-driven architecture, stream processing, Kafka, reactive systems, edge computing, scalability, latency minimization, asynchronous processing.*

Introduction. Traditional architectural approaches (such as monolithic servers with periodic batch processing of data) often do not meet the requirements of real-time processing. As a result, the industry has developed new architectural solutions and design patterns. These include microservices architecture, event-driven architecture, edge computing, the use of message queues and streaming platforms (Kafka, RabbitMQ, etc.), reactive systems, and others.

It is necessary to understand which architectural approaches and solutions enable the creation of applications capable of operating in real-time in a distributed digital environment (such as cloud and network conditions).

This article aims to:

- Analyze the main architectural patterns and technologies applicable to real-time systems;
- Review examples of real-world applications with specific latency requirements;
- Evaluate the advantages and limitations of various approaches (e.g., microservices vs monolithic in the context of real-time; cloud vs edge computing; use of asynchronous message exchange, etc.).

The relevance of this topic is driven by the rapid growth in data volumes and the demand for their immediate processing. According to a recent review, an increasing number of organizations are complementing traditional batch architectures with streaming architectures to process data "on the fly" [7]. Real-time architecture has become a necessary condition for competitiveness in many fields – for example, financial companies invest in low-latency infrastructure, while online services invest in ensuring instant user experiences. Therefore, the

question of how to correctly design the architecture of such systems is of significant interest both to the scientific community and to practicing architects.

Materials and Methods. For this review, English-language academic publications from open access [9] were used (e.g., the journal *Sensors* – architectures for IoT [6]). Specifically, the results of the following sources were applied: review articles on real-time stream processing and Big Data [1; 5] (e.g., DZone Refcardz 2023 [8]), industry reports [4] (such as the AWS technical blog on real-time analytics patterns [10]), as well as practical case studies described in the blogs of tech companies (Upsolver, Tinybird, etc.) [7; 11]. Significant attention was given to the Reactive Manifesto [2-3] and related materials, which define the principles for building high-load responsive systems.

The work was conducted as a systematic review of architectural solutions. A classification of the main approaches was created, which formed the basis for the structure of the results.

Results. Before discussing the architectures, it is important to clarify what is meant by real-time applications in the context of the digital (mainly network and cloud) environment. Unlike strict real-time systems (such as flight control systems, where there are strict deadlines in microseconds), most web and business applications belong to the class of soft real-time systems – they aim to minimize processing delays to ensure high responsiveness but do not have absolute deadlines. For example, for a user-facing web service, an interactive response within ~100 milliseconds is perceived as instantaneous. For a real-time analytics system, receiving insights in seconds instead of minutes already provides a business advantage [7]. Therefore, the goal of the architecture is to minimize latency (time from event/request to response) and ensure high throughput (to handle the stream of events). The main requirements for real-time application architectures are:

- Minimizing delays at all stages (input, processing, output). High-performance communication is used, and long-blocking operations are avoided.

- Concurrency and parallelism. Often, many events need to be processed simultaneously, so the architecture must be scalable and support parallel processing (multithreading, distribution across nodes).
- Resilience. Real-time services must operate continuously. A failure of one component should not stop the entire system – failure isolation and quick recovery are necessary [3].
- Elasticity. The load may increase dramatically (a surge in events), so the architecture must scale easily (horizontally in the cloud or using reserves).
- Data sequence. In some real-time systems (e.g., financial systems), it is important to correctly order events. The architecture must either guarantee ordering or be able to work without strict consistency (which is simpler for performance but creates logic complexity) [8].

Considering these requirements, several architectural approaches have emerged in the industry.

Microservices Architecture and its Role in Real-Time

Microservices is an architectural style where an application consists of numerous small, independent services, each of which performs a well-defined function, and they interact with each other through clear interfaces (most commonly network APIs) [4]. This approach became popular thanks to companies like Netflix and Amazon, who were the first to scale their monolithic applications into a set of microservices for better manageability. However, in addition to organizational advantages (division of development across teams), microservices also provide technical benefits for real-time systems.

They allow hot components to be scaled independently. For instance, if the service responsible for processing incoming events becomes a bottleneck, it can be scaled up by running more instances without affecting other parts of the system. This corresponds to the requirement for elasticity – resources are added dynamically to meet load [3].

Microservices often communicate asynchronously (through message queues, brokers), which supports resilience and low coupling. Instead of synchronously calling another module and waiting for a response (which creates delay and the risk of cascading failures), a service can send a message and continue working. This message-driven communication is one of the principles of reactive systems [3]. It improves decoupling of components and allows the system to remain responsive even when partial issues occur.

In a microservices architecture, it is easier to apply specialized optimizations for individual services. For example, a service responsible for caching frequently requested data can operate in memory (in-memory store) for instant access, while another service can write to a reliable storage. The functional separation facilitates targeted optimization to meet real-time requirements for specific parts.

However, microservices also have drawbacks when applied to real-time systems.

A call from one microservice to another over the network can take milliseconds, whereas a function call within a monolith takes tens of nanoseconds. For systems that require ultra-low latency (such as algorithmic trading), microservices can introduce unacceptable delays due to network calls. A practical solution is to minimize the number of hops between services along the critical path. For example, the critical data flow is processed within 1-2 services, rather than being routed through a dozen.

In a distributed microservices environment, performing ACID transactions across services is challenging. As a result, systems often need to shift to eventual consistency (where data is eventually synchronized). This is acceptable for many real-time systems (e.g., in an analytics system, instant consistency is not required, and it is acceptable if the data is updated after a few seconds). However, for applications like stock exchanges, more complex mechanisms need to be implemented.

Real-time microservices can generate huge streams of logs and events. Tracking where the delay occurs is non-trivial. Distributed tracing systems and queue monitoring need to be implemented, which increases infrastructure complexity.

Nevertheless, the experience of large companies shows that the benefits of microservices outweigh the challenges when implemented correctly. For example, Netflix processes trillions of events per day using hundreds of microservices – this architecture has proven capable of servicing millions of users simultaneously with minimal latency (video stream buffering, real-time recommendations, etc.).

Thus, microservices architecture supports real-time operations through scalability and component isolation but requires thoughtful interaction design (such as asynchronous messaging and caching) and significant engineering support (monitoring) to meet latency requirements. In practice, microservices are often combined with other approaches – such as event-driven integration through streaming platforms, which will be discussed further.

Event-Driven Architecture and Stream Processing

Event-driven architecture (EDA) is an architectural pattern in which the interaction between components is built around message-based events. Application components generate events (for example, "user clicked a button," "sensor sent a new measurement," "transaction completed"), and they respond to events received from other components, often through an intermediary – an event bus or message broker. In the context of real-time systems, EDA plays a central role because it allows information to be processed immediately upon the occurrence of an event, without waiting for scheduled requests. The main elements of EDA (see Table 1) are:

Table 1

Key Components and Advantages of Event-Driven Architecture (EDA)

Component / Advantage	Description
Message queues / event streams	Infrastructure for transferring events from sources to subscribers. Brokers such as Apache Kafka, RabbitMQ, AWS Kinesis, and Google Pub/Sub are used. These are optimized for high throughput and minimal latency. Kafka, in particular, handles millions of messages per second.
Event processors	Independent services (often microservices) that respond to specific events. Each processor executes its own logic, such as generating receipts, fraud detection, updating recommendations, etc. These services work concurrently with the same data.
Loose coupling of components	Components do not interact directly – interaction occurs through events. If an event processor is unavailable, events are stored in queues, ensuring the continuity of other parts of the system.
Asynchrony	No blocking in the processing flow: when specific event handlers are busy, the system continues to accept new events.
Horizontal scalability	As load increases, new handlers subscribed to the same events are added. Load distribution occurs automatically, and this approach is especially effective when handling telemetry streams and other large-scale streaming data.
Event ordering and retention	Brokers (especially Kafka) store events in logs, allowing for delayed reading. This enables replaying streams for debugging or reprocessing, such as when updating algorithms. Logging also helps mitigate temporary load spikes, preventing data loss.

An example of event-driven architecture is a stock market data streaming analytics system: incoming stock quotes are published to a Kafka topic, from which several services read them concurrently – one calculates aggregates (e.g., moving averages), another checks trading strategies, and a third sends notifications about significant changes to clients. All these actions happen almost simultaneously with each new quote arrival, ensuring an "analyze-as-you-go" approach.

It is worth mentioning the Lambda and Kappa architectures – these are big data processing patterns (see Figure 1).

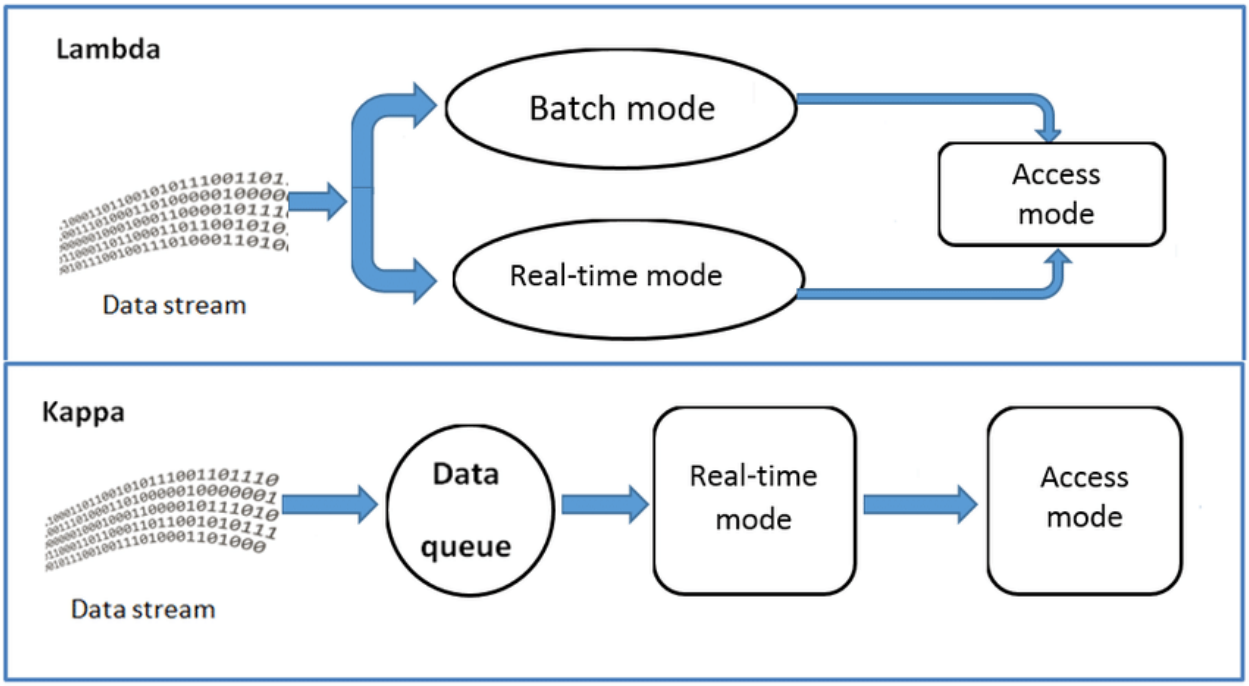

Fig. 1. Difference between Kappa and Lambda architectures [1]

The Lambda architecture splits processing into two paths: the batch layer (long-term batch processing for precise final data) and the speed layer (real-time stream processing with approximate data). This is an attempt to combine accuracy with performance. However, this scheme is complex, leading to the emergence of a simplified version – the Kappa architecture, where only stream processing is used (everything is treated as a continuous stream of events). In real-time applications, the Kappa architecture is more commonly used, especially with the advent of powerful stream analytics engines (e.g., Flink, Spark Structured Streaming), which can deliver results in a matter of seconds or milliseconds, previously requiring minute-long batch processing [1].

Limitations of EDA. The primary challenge is the complexity of development. Data resynchronization, the lack of global transactionality (each service works with its portion of events), and the complexity of debugging event sequences can all be problematic. Furthermore, event-driven systems require

thoughtful monitoring – it is important to track delays in queues and message "sticking." Tools such as Kafka Streams or Flink ease some of these challenges by providing high-level APIs for event processing (such as counters, time windows, etc.).

In general, event-driven architectures are the cornerstone of real-time systems, allowing for a natural description of reactive behavior: "received an event – processed it immediately, triggered new events," and so on. Many modern high-load systems are built around this paradigm.

Reactive Systems

The concept of Reactive Systems combines several principles, already discussed, into a unified development philosophy outlined in the Reactive Manifesto. According to this manifesto, a reactive system should be Responsive, Resilient, Elastic, and Message-Driven [2]. Essentially, this is a set of principles for building distributed real-time systems. The reactive approach emphasizes:

1. Asynchrony and non-blocking I/O. The use of tools like reactive programming libraries (RxJava, Reactor) enables writing code that does not block threads while waiting for results, instead using callbacks and completion events. This increases the number of concurrent operations without increasing the number of threads. For example, a web server on a reactive stack (Vert.x, Akka) can handle more requests on a single core than a traditional server with threads per request.

2. Component isolation and actors. A popular implementation of the reactive approach is the actor model, as seen in the Akka Toolkit. Actors are objects that communicate only by sending messages and process them sequentially, one at a time. This guarantees no concurrent conflicts within the actor, improving reliability. Actors can be easily distributed across nodes and support transparent recovery (Supervision strategies). This model greatly simplifies building fault-tolerant systems: if an actor fails, its parent restarts a new one without causing a global system crash. Many real-time telecom systems (e.g.,

WhatsApp, based on the Erlang/OTP actor model) have proven enormous scalability (millions of simultaneous connected users) with this architecture.

3. Back-pressure. Reactive systems include mechanisms to prevent overload: when a consumer cannot process events fast enough, it signals the source to reduce the rate. This is important to ensure that the system doesn't get overwhelmed during a flood of events. In a reactive stack (e.g., Reactive Streams specification in Java), back-pressure is built-in – consumers request a specific number of elements from the source, no more.

An example of a reactive system is Netflix's request-processing system. They used RxJava to orchestrate multiple external calls (to recommendation services, ratings, and databases) while forming the user's homepage. The reactive code allowed the system to collect data asynchronously with minimal delay, and if any service was slow, the system didn't completely block, but could partially display the interface. Another example is multiplayer online games. In a multiplayer game, it is essential that player actions are sent to others almost instantly. A reactive architecture based on actors can implement "rooms" (game sessions) as actors that receive events from players, update the game state, and send events to all participants. This approach scales well to hundreds of thousands of rooms with minimal latency within each one.

Limitations of the Reactive Approach: It is relatively complex for developers used to sequential programming. Debugging asynchronous code is harder. Also, a strictly reactive approach (e.g., requiring everything to be done via messages) is not always justified for simple applications – it can unnecessarily complicate the system. Therefore, a hybrid approach is often chosen: critical paths are made reactive, while non-critical components can be implemented more simply.

In conclusion, reactive architectures are more of a set of principles and best practices that help build real-time systems that meet the requirements of reliability and scalability. In practice, their implementation may be based on

microservices and events, but with the use of reactive frameworks and tools that ensure non-blocking operations.

Distributed Processing at the Edge

The classical approach of sending all data to the cloud or a data center can lead to unacceptable delays, especially when data sources are far away or when instant reactions are required (e.g., equipment control). Edge computing is an architectural solution where part of the computation is moved as close as possible to the data source, to the "edge" of the network. These can be local servers within the same network as the sensors, or directly smart devices/gateways capable of performing computations. Edge architecture reduces network latency. For example, in an industrial facility, machine sensors are connected to a local edge server, and vibration analysis is performed on it in real-time. Only the results or significant events are sent to the cloud for long-term storage. If all the data were sent to the cloud, the transmission + processing delay could be more than what is acceptable for preventing an accident. In fact, researchers note that for time-critical IIoT (Industrial Internet of Things) applications, the use of edge components is a necessary condition; otherwise, even small network delays can lead to critical situations [6; 9].

Architecturally, edge computing typically works in tandem with the cloud, forming a multi-tier architecture (see Figure 2):

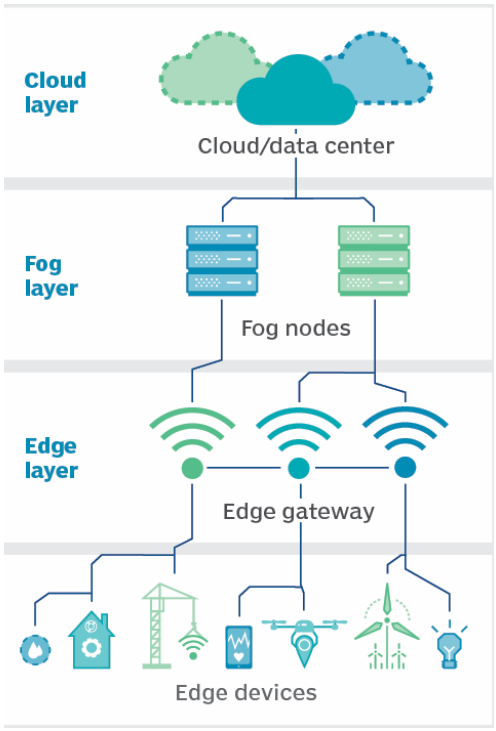

Fig. 2. Edge-to-Cloud Architecture Layers [5]

In Figure 2, the data flows are indicated by arrows: most of the data is processed at the middle layer (closer to the source), and only a portion is transmitted to the cloud. This architecture reduces latency and unloads the main network channels (Image: Psenda38, CC0). Edge architecture is widely used in 5G/6G networks, where the concept of Multi-access Edge Computing (MEC) involves placing servers directly within the telecommunications operator's infrastructure, near base stations. This allows, for example, streaming gaming services to place game servers closer to the player, achieving minimal ping. In 5G deployments, operators are collaborating with cloud companies to provide edge computing as a service.

Limitations of the Edge Approach: The need to deploy and maintain a large number of distributed nodes (which is more complex than simply keeping everything in one data center), potential security issues (since many nodes on the perimeter need protection), and the fact that edge often has limited computational resources (you can't infinitely increase algorithm complexity on a small node). Therefore, the architectural solution typically is as follows: critical parts of the

logic (such as pre-filtering, simple ML models for event detection) reside at the edge, while heavy processing (complex analytics, model training) is done in the cloud, where there are no resource limitations [6].

Examples of Architectures and Their Effectiveness

To link the discussed solutions with practice, let's consider several examples of real-time applications and the architectural solutions applied (see Table 2):

Table 2

Examples of real-time applications and the applied architectural solutions

Example	Requirement	Solution	Architecture / Technologies	Result
Fraud monitoring in a bank	Check each transaction for fraud in milliseconds before approval	Transaction stream via Kafka, parallel fraud checks by microservices using different rules	Event-driven pipeline, microservices, in-memory session store	<50 ms per transaction, fault tolerance
Social network / news platform	Immediate delivery of new posts to millions of subscribers	Fan-out events via queue, shard subscribers across servers	Pub-sub model, distributed queues (e.g., HDFS + Earlybird), load balancing	Delivery delay of hundreds of ms at high volumes
Cryptocurrency exchange	Process hundreds of thousands of requests per second with minimal delay	Actor model: one actor per trading instrument, order = message	Actors on a cluster, pub-sub via broker (e.g., NATS.io), in-memory processing	Sequence and execution speed (within ms)

Source: compiled by the author based on their own research

These examples show that combining multiple approaches is common. Microservices and actors can be combined with event-driven connectivity; edge computing can work alongside cloud streaming. A real-time system architect must combine tools based on the best fit for the specific task's requirements. A clear illustration of the value of real-time approaches is the "data value vs time"

graph (Figure 3). It shows that immediately after generation, data holds the highest value for business (for example, knowing that a user is currently viewing a product is highly valuable for showing relevant ads), but over time, its value exponentially decreases [11]. Real-time architectures allow benefits to be extracted while the data is "hot." This principle drives investments in the architectural solutions described.

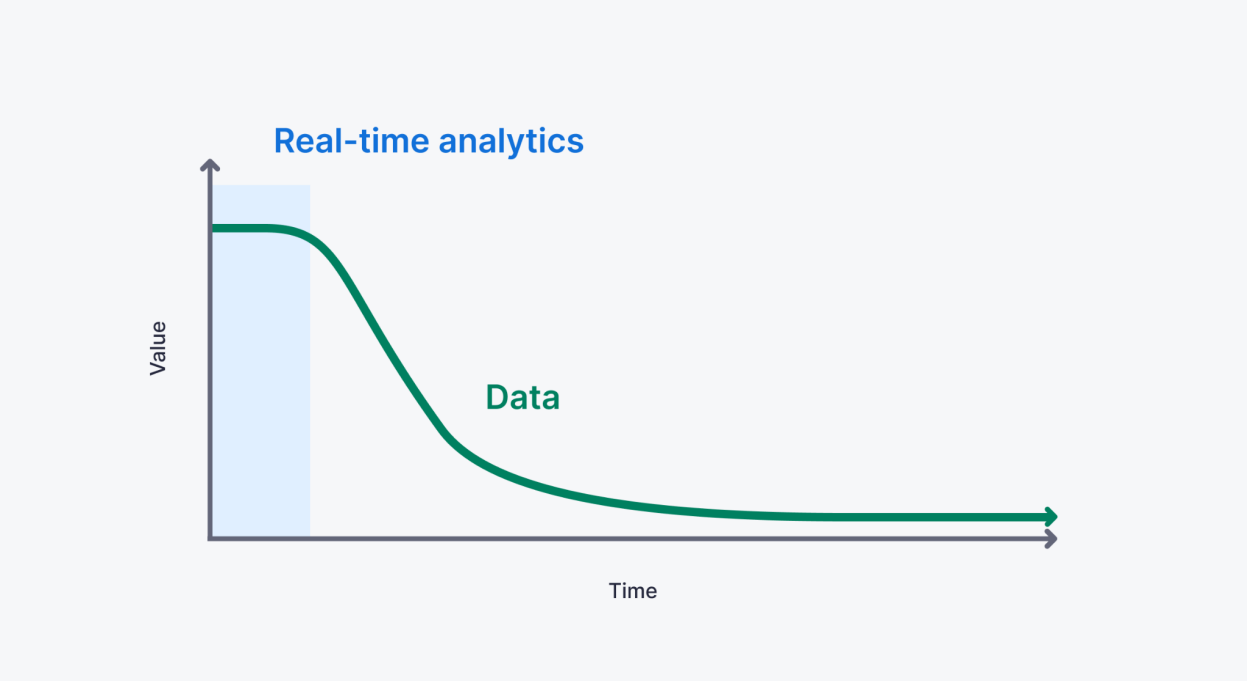

Fig. 3. Schematic representation of the decline in data value over time [11]

The green curve shows that data is most valuable immediately after it is generated, and then the value of the information quickly declines. The area on the left (highlighted in blue) represents the time interval during which real-time analytics can extract maximum benefit, before the data "cools down." Real-time architectures are aimed at processing data during this critical early phase.

Conclusion. Architectural decisions are a key factor in the successful implementation of real-time applications. The conducted review allows for the following conclusions:

1. Asynchronous distributed architectures dominate in digital real-time systems. The microservices approach, combined with event-driven

communications, has proven effective in ensuring low latency and high scalability. Such systems can remain responsive even under heavy loads due to parallel processing and component isolation.

2. Reactive principles (Responsive, Resilient, Elastic, Message-Driven) provide guidelines for design. In practice, this means: using queues and non-blocking I/O, implementing fault tolerance mechanisms (replication, automatic service restart), horizontal scaling based on load, and service interactions through asynchronous messaging. Implementing these principles (e.g., the Akka actor model) enables the creation of systems that continue to function correctly despite node failures and can easily scale to handle increased event traffic.

3. Edge computing is an important component of real-time solutions when minimizing network latency or ensuring local autonomy is required. Offloading part of the computation to the edge of the network (at factories, devices, or 5G MEC nodes) significantly reduces response time for critical applications (industrial control, autopilots, etc.). This architectural solution is recommended when the "physics" of data transmission (light speed, network hops) becomes a limiting factor.

4. Compromises are inevitable. Distributed real-time systems must balance data consistency and availability (according to the CAP theorem). A practical takeaway for developers is to identify in advance where strict consistency can be sacrificed for performance. For example, caching data on different nodes may lead to discrepancies of a few milliseconds – is this acceptable? For most user applications, yes; for financial calculations, probably not. The architecture should account for these requirements.

Examples have shown the effectiveness of patterns. Thanks to event-driven architecture, financial systems can detect fraud within tens of milliseconds, while social networks can spread messages almost instantaneously around the world. This has resulted in a significant advantage: companies can make decisions and

respond "on the freshest data," improving service quality and gaining competitive advantages (e.g., a more personalized user experience based on current actions, not yesterday's).

Practical recommendations: When designing a real-time application, one should: analyze the nature of the load (constant flow vs. event bursts) and choose an architecture capable of scaling dynamically (microservices + queue); identify components that are critical for latency – place them as close to the data source as possible and implement them with the fastest technologies (e.g., C++ service on the edge); use ready-made high-performance solutions: distributed streaming platforms (Kafka, Pulsar) for event exchange, in-memory data grids (Redis, Hazelcast) for caching hot data, asynchronous web servers for the frontend; integrate monitoring and back-pressure from the start. Without this, there is a risk that the system will only perform well up to a certain load threshold and then enter a degradation mode. A well-designed architecture ensures smooth degradation (disabling secondary features) instead of a crash.

References

1. Bobulski, Janusz & Kubanek, Mariusz. (2020). Big Data System for Medical Images Analysis. DOI: 10.20944/preprints202005.0274.v1.

2. Aceto, Luca & Attard, Duncan Paul & Francalanza, Adrian & Ingólfsdóttir, Anna. (2024). Runtime Instrumentation for Reactive Components (Extended Version). DOI: 10.48550/arXiv.2406.19904.

3. Amazon Web Services. Reactive Systems on AWS. – 2025. – URL: https://docs.aws.amazon.com/pdfs/whitepapers/latest/reactive-systems-on-aws/reactive-systems-on-aws.pdf (accessed: 07.04.2025)

4. Bhattacharjee, S. Microservices architecture and design: A complete overview. 2024. URL: https://vfunction.com/blog/microservices-architecture-guide/ (accessed: 07.04.2025).

5. Bigelow, S. J. What is edge computing? Everything you need to know. 2021. URL: https://www.techtarget.com/searchdatacenter/definition/edge-computing (accessed: 07.04.2025).

6. El Akhdar, A. et al. Exploring the Potential of Microservices in Internet of Things: A Systematic Review of Security and Prospects. *Sensors*. 2024. Vol. 24, no. 20. Article 6771. DOI: 10.3390/s24206771.

7. Franklin, J. Building a Real-Time Architecture: 8 Key Considerations. URL: https://www.upsolver.com/blog/building-a-real-time-architecture-8-key-considerations (date of access: 07.04.2025).

8. Garcia, M. Real-Time Data Architecture Patterns. URL: https://dzone.com/refcardz/real-time-data-architecture-patterns (date of access: 07.04.2025).

9. Kiangala, S., Wang, Z. An Effective Communication Prototype for Time-Critical IIoT Manufacturing Factories Using Zero-Loss Redundancy Protocols, Time-Sensitive Networking, and Edge-Computing in an Industry 4.0 Environment. *Processes*. 2021. Vol. 9, no. 11. Article 2084. DOI: 10.3390/pr9112084.

10. Sodabathina, R., Ly, B., Zuo, H., Radhakrishnan, S. Architectural patterns for real-time analytics using Amazon Kinesis Data Streams, part 1. 2024. URL: https://aws.amazon.com/ru/blogs/big-data/architectural-patterns-for-real-time-analytics-using-amazon-kinesis-data-streams-part-1/ (date of access: 07.04.2025).

11. Tinybird team. Real-time streaming data architectures that scale. 2023. URL: https://www.tinybird.co/blog-posts/real-time-streaming-data-architectures-that-scale (accessed: 07.04.2025).