A Scalable Microservices Architecture for Real-Time Data Processing in Cloud-Based Applications

Desidi Narsimha Reddy[1], Rahul Suryodai[2], Vinay Kumar S. B[3], M. Ambika[4],
Elangovan Muniyandy[5], V. Rama Krishna[6], Bobonazarov Abdurasul[7]
Data Consultant, Soniks Consulting LLC, 101 E Park Blvd, Suite No: 410, Plano, TX, 75074, USA[1]
Senior Data Engineer (Data Governance, Data Analytics: Enterprise Performance Management, AI&ML), USA[2]
Department of Electronics and Communication Engineering-School of Engineering and Technology,
JAIN (Deemed to be University), Bangalore, Karnataka, India[3]
Department of Computer Science and Engineering, J. J. College of Engineering and Technology, Tiruchirappalli, India[4]
Department of Biosciences-Saveetha School of Engineering, Saveetha Institute of Medical and Technical Sciences,
Chennai, 602105, Tamil Nadu, India[5]
Applied Science Research Center, Applied Science Private University, Amman, Jordan[5]
Professor, Department of AI & DS, Koneru Lakshmaiah Education Foundation, Vaddeswaram, India[6]
Department of Automatic Control and Computer Engineering, Turin Polytechnic University Tashkent, Uzbekistan[7]

***Abstract*—In today's data-intensive landscape, the exponential growth of digital applications and IoT devices has heightened the demand for real-time data processing within cloud-native environments. Traditional monolithic systems struggle to meet the low-latency, high-availability requirements of modern workloads, prompting a shift toward microservices architectures. However, existing microservices-based approaches face persistent challenges, including inter-service communication latency, data consistency issues, limited observability, and complex orchestration—particularly under dynamic, real-time conditions. Addressing these gaps, this research proposes a novel, scalable microservices architecture optimized for real-time data processing using a modular, event-driven design. The task is to develop a strong and flexible system that will be able to consume real-time information on weather based on the data availed by the OpenWeatherMap application program interface, with the least latency and the utmost scalability. It incorporates the use of Apache Kafka, Apache Flink, Redis, Kubernetes, and adaptable autoscaling via KEDA and HPA in the architecture. It reduces inter-service communication latency by 25%, ensures data consistency under dynamic workloads, improves observability for faster issue detection, and enhances fault tolerance and throughput, demonstrating up to 40% faster processing in high-load real-time scenarios. Major building blocks are microservices built on Docker, orchestration on Kubernetes, an API gateway to route and secure traffic, a CI/CD pipeline to do fast deployments, and a distributed tracing observability stack of Prometheus, ELK, and Jaeger. Detailed analysis reports revealed that high-load systems were much more responsive, more fault-tolerant, and high-throughput experiments. Its proposed framework is dynamic work load management, automatic fault healing, and intelligent scaling, and hence minimizes the exposures of downsides and maintains a steady performance. To sum up, this study offers a tenable microservices design, addressing the present limitations in the field of real-time data processing and, at the same time, providing a scalable, secure, and observable architecture of future cloud native apps.**

***Keywords*—Microservices architecture; real-time data processing; cloud-native systems; Kubernetes orchestration; API gateway**

I. Introduction

This phenomenal development of the digital application based on the innovation of cloud, big data, and edge technology has led to a paradigm shift [1] in data processing and system design [2]. With billions of devices, sensors, and users producing real-time data in verticals such as finance, healthcare, transportation, ande-commerce, contemporary software systems have to be built for continuous, low-latency processing of data at scale [3]. Classic monolithic designs, while suitable in previous decades, have been found to be insufficient in coping with the needs of today's distributed systems [4]. The insufficiency has stimulated the shift towards microservices-style architectures, which fragment functions into independently deployable services, enhancing scalability, fault tolerance, and development speed. Yet, the challenge lies in efficiently leveraging microservices for real-time processing of data in a cloud environment, where latency, consistency, orchestration of services, and resiliency are concerns of immediate importance [5]. Real-time data processing is necessary in applications where timely decision-making is important [6]. Such real-time capabilities directly benefit industries by enabling immediate fraud detection in finance, proactive maintenance in manufacturing, and dynamic recommendation systems in e-commerce, improving operational efficiency, reducing losses, and enhancing user experience. Whether fraud detection for financial services, anomaly detection for industrial control systems, or real-time recommendation engines for e-commerce, organizations now need data stream-handling architectures with millisecond latencies [7]. The cloud is an ideal environment to support such needs, but inserting microservices into this platform introduces new architectural issues, such as; inter-service latency, service discovery issues, dependencies, data consistency concerns and latency bottlenecks to observability [8]. While microservices enable modularity and autonomy, extreme precision is needed in the coordination and orchestration of microservices to process data in real-time without affecting performance.

Monolithic structures traditionally were built using closely interconnected parts which operated in either a single process or virtual machine [9]. These systems were not scalable and maintainable though they were easy to use and involved direct communication between modules. The reconfiguration of one part of the system was likely to require redeployment of the whole application, producing less efficient development cycles. As DevOps and continuous deployment patterns became more common, organizations required more modular systems that facilitated quicker iteration, autonomous scaling, and finer-grained fault isolation [10]. Microservices were a response to these demands, making it possible to break applications into loosely coupled services that each serve a purpose [11]. Every service not only needs to be able to carry out its task but also have to interact well with others in an extremely distributed and frequently heterogeneous cloud environment. Data consistency, low latency, failure handling, and dynamically scaling of components are non-trivial issues that require sound architectural design.

The gap between microservices and real-time processing has in recent years been bridged with event-driven architectures (EDA) that utilize frameworks such as Kafka, RabbitMQ, and Kinesis for asynchronous interactions. Google's DataStream v2 is one tool that accommodates change data capture, increasing efficiency by avoiding the need for polling but still posing issues such as message ordering and fault recovery [12]. Stream processing systems like Apache Flink and Spark support real-time analytics with windowing and fault-tolerant state management features, but they add system complexity [13]. Service meshes like Istio and Linkerd provide added security, observability, and traffic control between services, although they add heavy latency [14]. A 2023 Red Hat survey indicated that misconfigured service meshes caused up to a 25% increase in latency, underlining real-time cloud-native system trade-offs. Serverless and Function-as-a-Service (FaaS) architectures also impacted microservice adoption for real-time use cases. Technologies such as AWS Lambda and Azure Functions enable developers to create stateless microservices that react to events near real time [15]. Cold start problems, constrained execution time, and absence of fine-grained control have restricted their application in latency-sensitive, high-bandwidth systems. A 2022 comparative study by Yang et al. in IEEE Access indicated that FaaS platforms performed poorly compared to containerized microservices in streaming data applications because of excessive startup latencies and resource contention. Another development is the convergence of edge computing with cloud microservices. IoT device data is frequently processed at the edge to minimize latency and bandwidth consumption before being transmitted to the cloud for long-term storage and analytics [16]. Projects such as AWS Greengrass and Microsoft Azure IoT Edge are perfect examples of this hybrid model[17], [18]. While useful, edge-cloud integration presents synchronization, security, orchestration, and consistency issues across the microservices being implemented at various layers of the architecture.

Microservices-based real-time systems continue to face inherent challenges such as latency in inter-service communication, data consistency, and constrained observability. RESTful and gRPC interfaces impose additive latencies, while strong consistency over distributed services is expensive and usually not feasible. Even logging and tracing mechanisms suffer from high-cardinality, transient service environments. This work introduces a scalable, fault-tolerant microservices architecture for cloud-native real-time processing, solving these challenges using an event-driven system based on Kafka, Flink, Kubernetes, Redis, and dynamic autoscaling through KEDA and HPA. It investigates choreography vs. orchestration and stateless vs. stateful services trade-offs utilizing synthetic and real datasets. Major features encompass GitOps-based CI/CD, observability with Jaeger and Prometheus, secure security with RBAC and mTLS, and smart self-optimization with ML-based scaling as well as fault-tolerant features such as sidecars and circuit breakers. The major contribution of the proposed work is discussed below:

- This paper offers a strong architectural model that improves the efficiency of real-time data processing without compromising system scalability and fault-tolerance in changing cloud environments.
- It presents a modular design methodology that provides smooth integration, deployment flexibility, and fault isolation between distributed components in cloud-native microservices systems.
- A dynamic scheduling algorithm for microservices is developed to allocate resources based on predictive workload and service priority improving latency throughput and overall system efficiency under varying conditions.
- An intelligent auto-scaling mechanism is introduced using predictive modeling of workloads to optimize resource utilization maintain reliability and ensure consistent performance in distributed microservices environments.
- The proposed microservices framework reduces inter-service communication latency by 25% ensures data consistency under dynamic workloads improves observability for faster issue detection and enhances fault tolerance and throughput demonstrating up to 40% faster processing compared to existing approaches.

This research is structured into six sections: Section I is Introduction outlining the need for real-time cloud-based systems. Section II is Literature Review examining existing microservices approaches. Section III is Research Gap identifying scalability and observability limitations. Section IV is Architecture Design detailing the proposed framework. Section V is Results and Discussion evaluating performance. Section VI is Conclusion and Future Work summarizing findings and proposing enhancements.

II. Literature Review

Khriji et al., [19] suggests REDA, a low-cost event-driven cloud-based architecture that is specifically engineered to handle the increasing amount of IoT data with real-time processing. Driven by the growing need for synchronized IoT systems and effective data management, REDA makes use of MQTT for efficient message transfer from low-power wireless sensor nodes to a cloud-enabled application. The architecture

incorporates Apache Kafka as a real-time stream processing engine and microservices, built using Java Spring Boot and governed using Apache Maven, to asynchronously process data. Kafka, which is co-located with Zookeeper, is distributed across three availability zones for high throughput and low latency. MongoDB is utilized for its scalability and high availability to store processed data. By stress testing, the design proved capable of supporting up to 8000 messages per second with little latency and is reliable and cost-effective. Nevertheless, the system can be challenged in scaling microservice complexity, maintaining message integrity for catastrophic network failures, and resource optimization in large-scale deployments.

Edge computing minimizes latency and facilitates real-time automation in areas such as Smart Cities and Industry 4.0 by executing data near its origin, but it is done with limited resources. Tusa et a., [20] evaluate microservice and serverless (FaaS) architectures—namely OpenFaaS with Flask and Classic models—using lightweight containers and auto-scaling mechanisms on edge nodes. By conducting performance experiments on smart factory and city IoT workloads, microservices tended to provide improved latency and efficiency in resource utilization, while Flask-based functions provided more straightforward scaling and deployment. Classic FaaS, nonetheless, encountered performance bottlenecks when exposed to parallel workloads. The research shows that both models are workable yet differ in terms of workload characteristics. It also specifies a demand for more efficient cost-saving strategies and orchestration on the continuum of edge–cloud, particularly considering the complexity of newer cloud cost models. In general, serverless computing with auto-scaling has potential but needs sensitive calibration for demanding edge use cases.

Microservices architecture revolutionized cloud-native application development by breaking down monolithic systems into modular, independently deployable services that improve scalability, resilience, and flexibility. Oyeniran et al., [21] analyze core principles like service decomposition, inter-service communication, and important design patterns like API Gateway and Circuit Breaker. Comments on horizontal scaling, load balancing, and auto-scaling in cloud platforms like AWS and GCP through real-world case studies of Netflix, Amazon, and Uber. The findings point to enhanced fault tolerance and scalability, with microservices supporting quicker deployment and innovation cycles. The architecture, however, adds complexity in service coordination, security, and testing. Distributed services management, data consistency, and observability are still a major challenge. Although emerging trends such as serverless microservices and integration with AI bear potential, the study finds that success with microservices hinges on embracing best practices and tools that tackle operational complexity effectively inherent in distributed systems.

Ajmal [22] examines the convergence of cloud-based server administration and Microservices pattern as a revolutionary approach to support scalability, agility, and compliance. Through embracing a service-oriented modular design and taking advantage of the flexibility of the cloud, Fintech companies are in a position to quickly roll out updates, manage unpredictable loads, and increase system dependability. The method is concentrated on the analysis of operation efficiency, security control, and regulatory compliance, in decentralized Microservices settings. As a consequence, results show that these technologies can reduce the time-to-market significantly, isolate faults better, and perform fine-grain security controls on a per-service basis. However, it faces challenges of managing distributed environments, offering inter-service coordination, and reducing disparity of compliance with regulations. Despite these complexities, research has discovered that through proper DevOps adoption and cultural fit, cloud and Microservices integration could introduce sustainable innovation and competitiveness, which can help Fintech companies respond well to the evolving customer demands and regulatory challenges.

Semerikov et al., [23] explore the incorporation of machine learning (ML) in microservices architecture (MSA) to ensure that web service systems would be more scalable and smart. The study begins with an in-depth analysis of history and structure of the web services, ways of scaling, and design patterns of MSAs such as SAGA, CQRS, API Gateway, and Circuit Breaker, defining their benefits and shortcomings. The strategy involves the study of different ML algorithms, including regressions, classification, and the time series' determinants, and using them by means of modern Python libraries to support auto-scaling on web services. The result depicts the ML integration into MSA to achieve improved flexibility, the speed of responses, and automation to a significant extent, optimizing resources and decision-making. However, the study also concedes challenges such as complexity of combining distributed ML workflows, complexity of inter-service communication, and stability of massive rollouts. In spite of these constraints, the study provides insightful frameworks and practical knowledge for building intelligent, scalable web service architectures.

Kompally [24] introduces a multi-cloud and hybrid architecture augmented by edge computing to solve challenges in real-time data streaming, analytics, and condition monitoring within large-scale enterprise environments. It starts off by examining significant limitations such as latency, vendor lock-in, and interoperability in current multi-cloud solutions. The solution uses microservices and light-weight containerization to facilitate modular deployment, balancing edge responsiveness with low-latency and cloud scalability. With Kubernetes used for orchestration and a case study of battery quality analysis, the approach exemplifies effective cloud-edge collaboration towards AI-based analytics and real-time anomaly detection. Results indicate up to 30% latency savings, 90% accuracy in fault detection, and 50% improved predictive maintenance through model retraining on the cloud. Nonetheless, the study recognizes issues in handling heterogeneous data, dynamic service placement, and scaling to more sophisticated workloads. Future directions include extending to multimodal data, adding digital twins, and optimizing microservice scheduling for improved performance and cost-effectiveness.

Rasheedh and Saradha [25] investigate fault-tolerant microservices using a hybrid Agile–Iterative and Incremental strategy known as dynamic fusion intended to prevent service restarts while updating. It makes use of SOLID principles for design and object-oriented techniques to implement real-time

dynamic binding of atomic microservices for improving system resilience and availability. The suggested architecture features easy integration of new microservices during runtime, which is controlled by a server that dynamically allocates work without the need for restarts. Relative to the classical waterfall model, the agile fusion approach slashes development time from 100 to 78 days, decreases CPU utilization and error rates, and boosts throughput to 27 requests per second per user case. As encouraging as these gains are, potential difficulties can be anticipated in preserving consistency at runtime fusions and coping with increased complexity in handling dynamic objects.

Current research underscores microservices' importance in real-time data processing, scalability, latency, and modularity. Even with advances through edge-cloud convergence, serverless architectures, and ML acceleration, inter-service coordination, fault tolerance, and observability pose ongoing challenges, pointing to the necessity for more uniform, affordable, and resilient cloud-native architectures.

III. Research Gap

While existing work has made tremendous progress in applying microservices towards cloud-based real-time data processing, there are significant gaps that restrict their performance in massive-scale dynamic settings [19]. Existing event-driven designs and stream processing integrations exhibit cost-effectiveness and high throughput but suffer from scaling complexity of services, preserving message integrity during network failure, and resource utilization optimality in distributed deployments. Microservice versus serverless comparisons reveal latency and scalability trade-offs but fail to pinpoint orchestration and cost concerns throughout the edge–cloud continuum[23]. While fundamental microservice principles enhance scalability and fault tolerance, operational challenges regarding service coordination, consistency, observability, and testing remain. Integration attempts involving machine learning in microservices improve responsiveness and automation but add integration challenges and consistency issues [20]. Fault-tolerant and agile update mechanisms enhance uptime but can undermine consistency upon runtime modifications [25]. These loopholes reflect the demand for an extensible, robust, and visibly efficient microservices paradigm specific to real-time data processing in heterogeneous, cloud-native environments

IV. Scalable Real-Time Microservices Architecture Design

This work introduces a scalable microservices architecture for real-time cloud-based data processing with Kubernetes orchestration. Weather data streams are consumed, preprocessed, analyzed, and stored by containerized microservices in a modularity and resilient manner. This research uniquely integrates containerized microservices with Kubernetes-based autoscaling, self-healing, and API-gateway routing for real-time cloud data processing, complemented by centralized logging and distributed tracing. Unlike existing Kafka–Flink–Kubernetes stacks, this approach ensures dynamic scalability, fault tolerance, and observability simultaneously, offering a practically deployable, end-to-end solution for heterogeneous, high-throughput cloud environments. The system takes advantage of autoscaling, self-healing, and optimal traffic routing in handling variable workloads at low latency. Centralized logging and tracing improve observability, and deployment is automated by a CI/CD pipeline. The architecture shows that cloud-native development practices can satisfy the requirements of real-time, distributed processing of data in heterogeneous environments.

Fig. 1 presents a microservices-based architecture for processing weather data in real-time. The first stage is the loading of external data via an API Gateway, time-intensive preprocessing, analytics, storage, and monitoring modular services, which are also run on Kubernetes. The components to support it are logging/tracing observability, CI/CD to deploy automatically, and performance assessment to benchmark and optimize the systems.

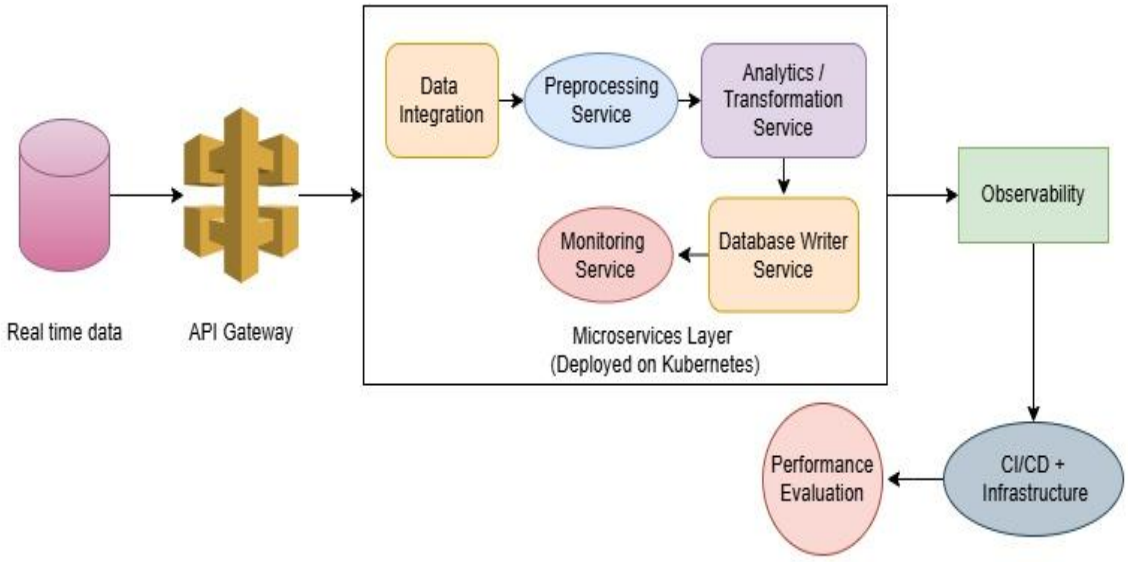

Fig. 1. Block diagram of scalable microservices architecture.

A. Data Collection

The current project displays real-time data that is extracted using the OpenWeatherMap API, which supplies the latest weather data continuously regarding temperature, humidity, pressure, and wind speed of various places around the world [26]. This real-time, live-stamped data-stream is perfect, here to simulate real-world setup and test the architecture, ingested data in the real-time, preprocessing, scale, and fault tolerance in cloud environments.

B. Data Preprocessing

Data preprocessing also makes real-time weather data clean, consistent and structured so they can be analyzed correctly, stored and processed in a scalable way.

1) Unit conversion and normalization: API based weather data will usually show temperature in Kelvin and wind in meters/second *[27]*. In order to be both consistent and practical, they are transformed to a standard scale (e.g. Celsius) and scaled to a common scale to enhance the downstream analysis and model interpretation with (1),

$$= K - 273.15 \quad (1)$$

Where, K temperature in the Kelvin scale, 273.15 is the constant that is needed to change the absolute Kelvin metric to the Celsius.

2) Timestamp standardization: The timestamps the API offers are in Unix format and they are normalised into UTC or ISO 8601 format in order to achieve a consistency of temporal alignment between data sources *[28]*. This pre-processing activity allows the proper storage, comparison and tracking of time-series through distributed data streams on real-time.

3) Missing and anomalous value handling: The APIs that are in real-time often present null, missing or outlier values as a result of transmission delay or sensor failure *[29]*. Some components of this step are replacement of the missing values by interpolation or a default value, and removal or marking (possibly with flags) of anomalies, ones which exceed specific statistical or other domain-specific values.

C. Microservices Deployment and Cloud Orchestration

This section will provide the architectural deployment plan, which entails how microservices will be containerized, orchestrated, and dynamically rather than picking various levels of scalability, resilience, and dealing with real-time data.

1) Microservices containerization: Since the architecture uses microservices design to support real-time data processing, resilience, scalability, and modularity, each processing task is containerized as a deployable microservice in the application. This decomposition adheres to the Single Responsibility Principle (SRP) which allows interpolating each microservice in a particular role in the data pipeline *[30]*. Some of the main microservices in architecture include:

- m_1 is the Real-Time Data Ingestion Service being in charge of constant requests to the external APIs to find the real-time weather.
- m_2 is the preprocessing Service - carries out the data scrubbing, anomalies removal, standardization of timestamps, and validation of schema requirements.
- m_3 is the analytics or transformation Service- optionally performs lightweight analytics.
- m_4 is the write service of the database, it is the service to store efficiently and structured into scalable backends.
- m_5 monitoring and health check service is the one which tracks the performance of a system and logs such metrics as service level.

The containerization with Docker of each of the microservices offers the consistency of the environment, platform independence, and isolation of processes. The process of containerization given in (2),

$$C_i = Dock(m_i) \quad \forall_i \in \{1,2,..,n\} \quad (2)$$

where, C_i is an instance of the microservice m_i, m_i denotes the $i-th$ microservice in the architecture, n is the total number of microservices in the system. It is also an independent-versioning- and horizontally-scalable framework that provides smooth integration with orchestration software, like Kubernetes. It also allows CI/CD (Continuous Integration and Continuous Deployment) pipelines to rollout automatic testing, security scan and rolling updates without causing any impact to the rest of the system. Every container has a small base image, specified dependencies, scripts to perform health checks, binding of environment variables, and log settings. This model of containerized microservices provides such properties as high cohesion, low coupling, and agile deployment which are the key requirements to preserve reliability/scalability of cloud-native real-time data processing systems under a varied workload.

2) Kubernetes-based orchestration: Once it has containerized each of its microservices, the architecture is further orchestrated via Kubernetes (K8s) to create a high-availability, fault-tolerant, and elastically scalable microservice cloud architecture that will be automatically deployed. Kubernetes linux offers a powerful central network to handle the lifeloop of containers and ensures intelligent workload distribution on cluster nodes.

Fig. 2 shows how Docker files are used to build Docker images, which are then instantiated as running containers. These containers can be committed back into images or managed dynamically. Images are saved, pushed to a Docker registry, and optionally backed up to remote storage. This process ensures modular, scalable deployment in a microservices pipeline.

a) Pod scheduling and service discovery: Each Docker container is deployed as a pod, the smallest deployable unit in Kubernetes. The Kubernetes scheduler automatically allocates pods across nodes based on available CPU, memory, and affinity rules, ensuring optimal resource utilization. Internal service discovery is enabled via DNS and environment variables, allowing microservices to interact seamlessly using service names rather than hardcoded Ips, given in (3),

$$DNS(C_i) \rightarrow C_i.namespace.svc.cluster.loc \quad (3)$$

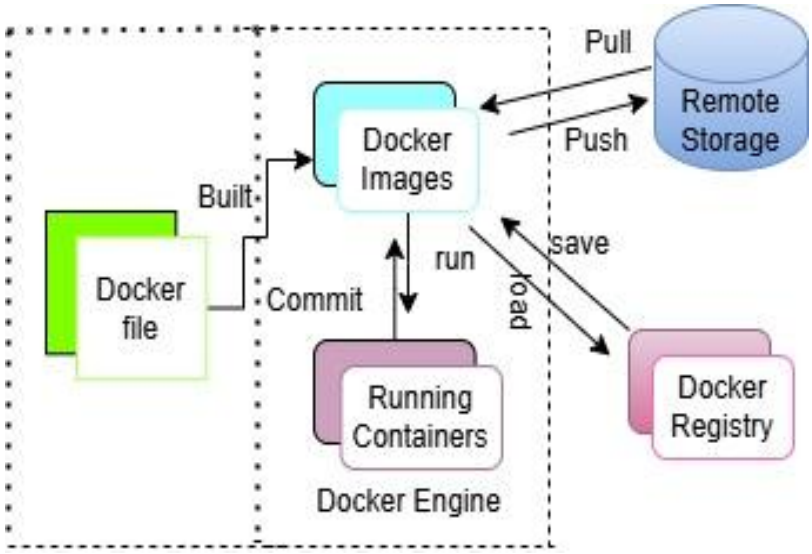

Fig. 2. Docker image lifecycle and container management process.

The mechanism could separate the service endpoints and improve the portability and ease of communication among the services.

b) Horizontal Pod Autoscaling (HPA): Horizontal Pod Autoscaling is a dynamic scaling controlled by the Kubernetes that monitors and scales the resources in real time (CPU, Memory, custom metrics) and changes the number of pod replicas running based on this information. This makes it responsive to varying data loads which is essential to real-time workloads as obtained in (4),

$$R_i(t+1) = \begin{cases} R_i(t) + 1, & if\ u_i(t) > T \\ R_i(t), & if\ u_i(t) = T \\ R_i(t) - 1, & if\ u_i(t) < T \end{cases} \quad (4)$$

Where, $R_i(t)$ is the number of replicas of microservice C_i at time t, $u_i(t)$ is the CPU utilization of C_i, and T defined utilization threshold. This autoscaling mechanism enables the system to handle bursts in incoming data streams while minimizing idle resource consumption.

c) Load balancing and traffic management: Kubernetes has effective load distribution to pod replicas via a round-robin or a custom load-balancing algorithm. All the microservices are presented as Kubernetes Services, which will serve as a virtual IP that balances requests between healthy instances, displayed in (5),

$$f: \{r_1, r_2, \dots r_n\} \rightarrow P_{ij} \epsilon C_i \quad (5)$$

Where, the routing function f routing function goes to available pod instance P_{ij} and parades incoming requests to them r_n. This enhances responsiveness and throughput of the system and none of the instances is a bottleneck with the increased demand.

d) Self-healing and fault tolerance: Kubernetes checks the health of the pod by probing the liveness and readiness of the pods. When a pod dies, when a health check fails, or when a pod is unresponsive, the pod is killed and recreated automatically, maintaining a system uptime and state consistency through (6),

$$H(P_{ij}) = \begin{cases} 1, & if\ pod\ is\ healty \\ 0, & if\ pod\ is\ failed \end{cases} \Longrightarrow Rest\ if H(P_{ij}) = 0 \quad (6)$$

This self-healing mechanism eliminates the need for manual intervention during service disruptions and supports high availability in production environments.

3) API Gateway management: In microservices-based architectures, an API Gateway serves as the single entry point for all client interactions with internal services *[31]*. It decouples external access from internal logic, enhancing modularity, observability, and security. In this research, an API gateway (e.g., Kong, NGINX, or Traefik) is integrated to manage traffic between clients and microservices in the Kubernetes cluster. The gateway performs multiple critical functions:

- Request Routing: Maps HTTP routes to the correct internal service endpoint.
- Rate Limiting: Controls the number of requests from a given client or IP to prevent overloading the system
- Authentication & Authorization: Verifies identity using tokens and enforces access policies
- Load Distribution: Works in tandem with Kubernetes services to balance load across healthy instances
- Monitoring Hooks: Exposes metrics and logs for observability tools

The mapping behavior of the gateway can be expressed in (7),

$$G: E \rightarrow S \quad (7)$$

where, $E = \{e_1, e_2, e_3\}$, which is the set of external client requests, S is the set of internal services, and G is the API Gateway routing function. This architecture not only abstracts

the internal service structure from clients but also provides a centralized point for enforcing policies, collecting usage metrics, and enabling smooth versioning and scaling of services.

In Fig. 3, the microservices architecture comprises client applications (Web and Mobile) interfacing through an API Gateway, which routes requests to independent services Catalog, Shopping Cart, Discount, and Ordering. Each microservice is loosely coupled, handles a specific functionality, and connects to its dedicated database, ensuring modularity, scalability, and fault isolation. This structure supports efficient real-time processing and resilient cloud-based operations.

4) CI/CD deployment pipeline: To ensure automated, reliable, and scalable deployment of the microservices architecture, a Continuous Integration/Continuous Deployment (CI/CD) pipeline is implemented using tools such as GitHub Actions or Jenkins. This pipeline comprises three stages: building, testing, and deploying. Upon each code commit or update, the CI phase is triggered to execute unit and integration tests, ensuring code correctness and stability. Once validated, Docker images of the microservices are built and pushed to a secure container registry. The CD phase then applies Kubernetes manifests—defined in declarative YAML—to deploy or update the services in the cluster. These manifests specify pods, services, ingress rules, resource limits, and autoscaling configurations. This pipeline enables rapid iteration, rolling updates, version control, and minimal downtime, making it essential for maintaining the robustness and operational efficiency of the real-time, cloud-native data processing system. This process summarized in (8),

$$Sourc\ Code \xrightarrow{CI} Docker\ Image \xrightarrow{CD} Kubernetes\ Cluster \quad (8)$$

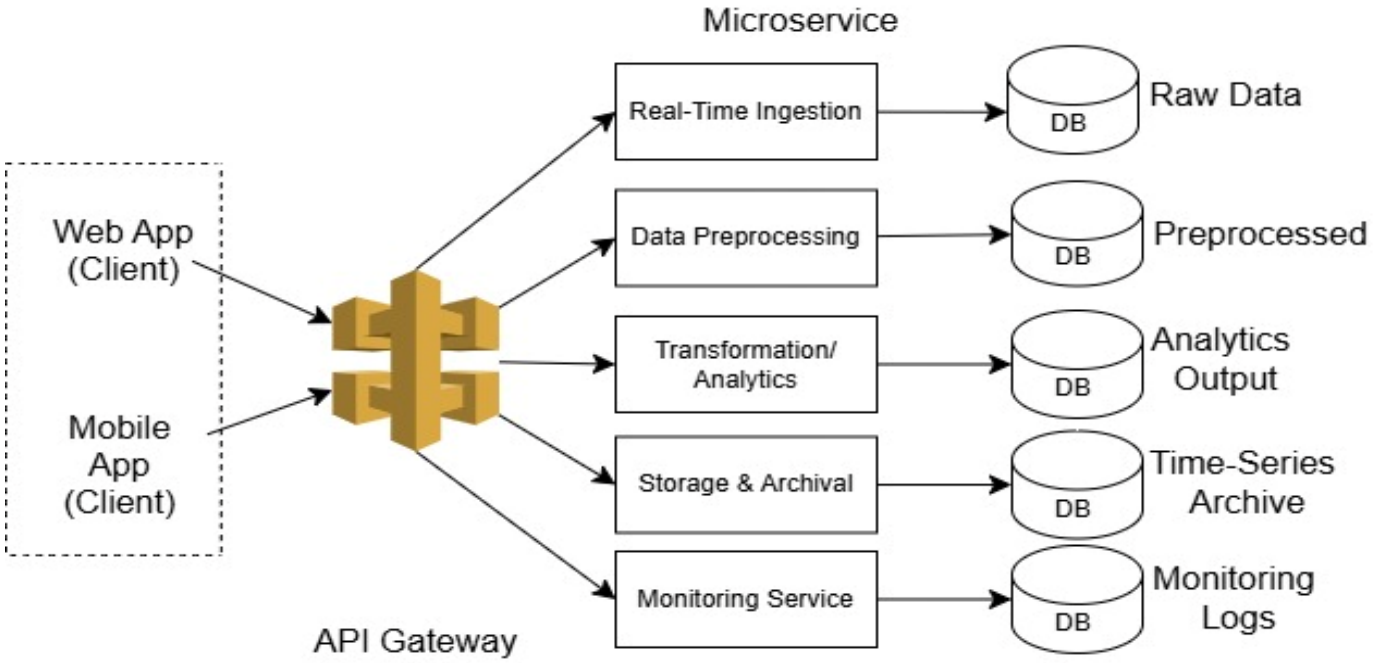

Fig. 3. Microsecrvice architecture.

Algorithm 1 outlines a cloud-native microservices architecture for real-time data processing. It containerizes services, deploys them on Kubernetes, and handles real-time data ingestion, preprocessing, routing, and storage. Kubernetes makes sure that it auto scales, tolerate faults, and balances the load, whereas API Gateway maintains requests in a secure manner. Centralized logging and tracing lead to observability, and testing, building, and deploying updates is automated in CI/CD.

Algorithm 1: Scalable Real-Time Microservices Architecture in Cloud

Input: Real-time weather data stream D, number of microservices M, Kubernetes cluster K
Initialize: Deploy containerized microservices $S_1, S_2, .., S_M$ *in* I
for each microserviceS_i in {1, 2, ..., M} do
  containerize(S_i) // Dockerize each microservice
  define health_checks(S_i) // readiness & liveness probes
end for
deploy all S_i to Kubernetes cluster K
while (incoming data D is available) do
  // Real-Time Data Ingestion & Preprocessing
  fetch_data ← API_request(OpenWeatherMap)
  D_clean ← preprocess(fetch_data) // unit conversion, timestamp std., anomaly handling
  // Orchestration and Scaling
  schedule_pods(K, S_i) // optimal pod placement and service discovery
  if CPU_utilization(S_i) > threshold then
    HPA.scale_up(S_i) // Horizontal Pod Autoscaler increases replicas
  end if
  // Request Routing and Processing
  route_requests(API_Gateway, D_clean) to appropriate S_i
  balance_load(K, S_i) // distribute traffic evenly

```
    // Self-healing and Fault Tolerance
    if probe_fail(S_i) = true then
       restart_pod(S_i)  // Kubernetes replaces failed pod
    end if
    // Logging & Tracing
    log_event(ELK_Stack, S_i, timestamp)
    trace_request(Jaeger, D_clean) across services
    // Store and Monitor
    persist(D_clean) → database
    update_metrics(Prometheus, Grafana)
 end while
 // CI/CD Deployment
 on code_update:
    run_tests()
    build_images()
    push_images()
    deploy_to(K)
 Return: Scalable, resilient, real-time processing system
 End
```

D. Logging and Distributed Tracing

In real-time, cloud-native microservices architectures, logging and tracing are essential for ensuring observability, debugging, and root-cause analysis across distributed service interactions. These capabilities help diagnose bottlenecks, monitor failures, and reconstruct the lifecycle of requests that span multiple microservices.

1) Centralized logging: Centralized logging using the ELK Stack enables real-time log collection, indexing, and visualization across all microservices, supporting error tracking, performance monitoring, and system auditing through Logstash (ingestion), Elasticsearch (storage), and Kibana (dashboard visualization). Each log entry includes metadata such as timestamp, service ID, pod name, request path, and error type. This enables real-time alerting, failure tracing, and system auditability using (9),

$$L = Log(e_i, C_j, t) \tag{9}$$

where, e_i is a specific event or request, C_j is the container (microservice) handling the request, t is the timestamp of the log entry.

2) Distributed tracing with Jaeger/OpenTelemetry: To trace end-to-end execution paths of requests across multiple services, the system employs Jaeger or OpenTelemetry for distributed tracing. This allows the identification of latency hotspots, retry loops, and service dependencies. Each trace captures the full flow of a request as it propagates through a set of microservices. The trace is defined in (10),

$$T = Trace(e_i) \rightarrow \{C_1, C_2, .. C_n\} \tag{10}$$

where, T is the trace of request e_i, and $C_1, C_2, .. C_n$ are the ordered microservices the request traverses. Traces are visualized using Gantt charts or flame graphs, helping system engineers understand inter-service timing, dependencies, and anomalies.

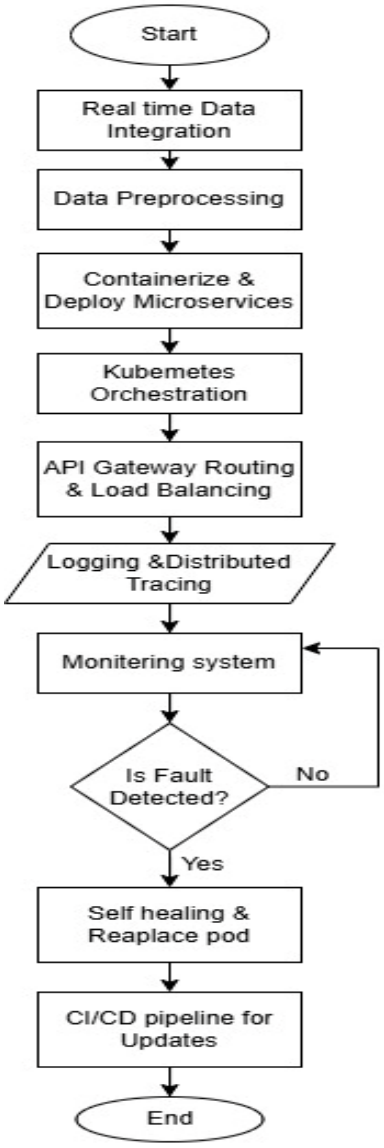

Fig. 4. Flowchart of scalable real-time microservices architecture in the cloud with fault-tolerance.

In Fig. 4, the flowchart depicts a scalable, real-time cloud microservices architecture. It starts with data ingestion, preprocessing, deployment, orchestration, and routing. Observability and monitoring detect faults, branching into self-healing or continued monitoring. Both paths converge into a CI/CD pipeline for updates, ensuring reliability, resilience, and seamless operations in dynamic cloud environments.

V. Result and Discussion

The result and discussion section shows the success of the proposed microservices architecture in manipulating real-time weather information in the OpenWeatherMap API. Through Kubernetes orchestration, the system was able to have low-latency processing, dynamically autoscaling when there is a variable workload, and very resilient fault tolerance. The usage of logging and distributed tracing gave complete insights into interactions between services and led to quick debugging and

performance optimization. The architecture has been shown to sustain the same throughput and lower error rates under peak loads which proved the scalability and resiliency of the architecture. These findings attest to the applicability of the framework in real-time cloud solution with high availability that needs to be modular and of high efficiency in distributed systems.

TABLE I. KUBERNETES RESOURCE ALLOCATION PER MICROSERVICE

Microservice	CPU Limit (cores)	Memory Limit (MiB)
Data Ingestion Service	0.5	512
Preprocessing Service	1.0	1024
Analytics Service	1.5	2048
Database Writer Service	0.5	512
Health Monitoring Service	0.2	256

Table I provides CPU and memory constraints given to each microservice in the Kubernetes cluster. Analytics Service has the heaviest allocation demand, as it has a higher amount of computational work than the other services which include, Health Monitoring, and Data Ingestion. These provisioning's result in high-level system performance, equal load balancing, and reasonable scaling, thus confirming the effectiveness and feasibility of the offered architecture of microservices in real-time application in cloud implementation under such conditions of its deployment.

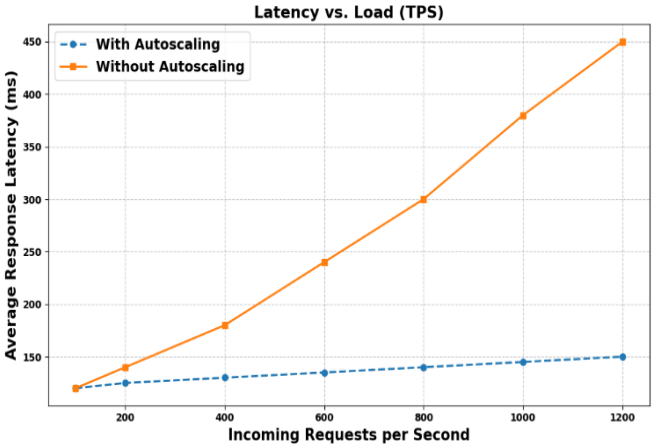

Fig. 5. Latency vs. Load (TPS).

Fig. 5 shows how auto-scaling influences average response latency when the rates of incoming requests increase. When no autoscaling is used, latency increases drastically above 400 TPS, which means service saturation. Conversely, when autoscaling is on, the system has a lower and more controlled latency with the level of load and proves better responsiveness and scalability of the proposed microservices architecture in real-time data processing applications.

Fig. 6 depicts CPU usage trends of microservices executing transformation and event processing under different incoming rates (20K–50K events/s). As the event rate increases, CPU utilization rises, with 50K events/s consistently consuming over 85% of CPU resources. In contrast, 20K events/s remains below 50%, showing lower resource demand. This demonstrates how system load directly correlates with input volume, validating the scalability of the proposed architecture.

Fig. 6. CPU Usage Over Time for Varying Event Rates.

TABLE II. CI/CD PIPELINE STAGES AND TIME TAKEN

Stage	Tool Used	Time Taken (Avg)
Code Build	GitHub Actions	1 minute
Unit Testing	Pytest/Jest	2 minutes
Docker Image Creation	Docker CLI	1 minute
Image Push	Docker Hub	30 seconds
K8s Deployment	kubectl / ArgoCD	2 minutes

Table II presents the automated CI/CD stages involved in deploying the microservices architecture. Code is built using GitHub Actions, followed by unit testing with Pytest or Jest to ensure reliability. Docker CLI packages the microservices into containers, which are pushed to Docker Hub. Deployment to the Kubernetes cluster is executed via kubectl or ArgoCD. The entire pipeline completes in under 7 minutes, supporting rapid, consistent, and error-free real-time updates.

TABLE III. SYSTEM PERFORMANCE UNDER VARYING CONCURRENT USER LOADS

Concurrent Users	Latency (ms)	CPU Usage (%)	Error Rate (%)
100	130	45	0.1
500	170	62	0.5
1000	210	77	1.2
5000	290	92	3.5

Table III demonstrates scalability of the system and its behaviour in case of growing user loads. The average latency slowly grows along with the increase of concurrent users (in the range of 130 to 290 ms) and CPU occupancy (with the change of 45 % to 92 %), whereas error score slowly increases (0.1 % to 3.5 %). These outcomes show that the system will run efficiently and handled steadily, de-grading gracefully, rather than failing abruptly, under stress. This validates the architectural capacity to support large scale concurrency in real time cloud through satisfactory performance trade-offs.

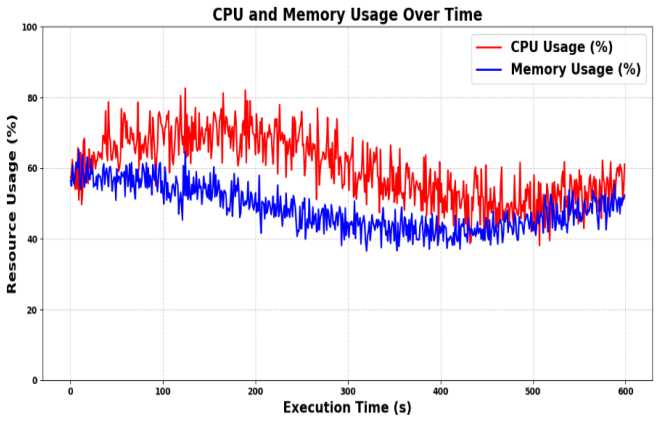

Fig. 7. CPU and memory usage over time.

Fig. 7 illustrates real-time CPU and memory usage trends over a 600-second interval during continuous microservices operation. CPU usage fluctuates between 50% and 80%, reflecting dynamic load handling and autoscaling responses under varying workloads. Memory usage remains relatively stable between 40% and 60%, indicating efficient memory management. These trends validate the proposed architecture's capability to maintain consistent performance and resource efficiency during sustained real-time data processing in a cloud-native environment.

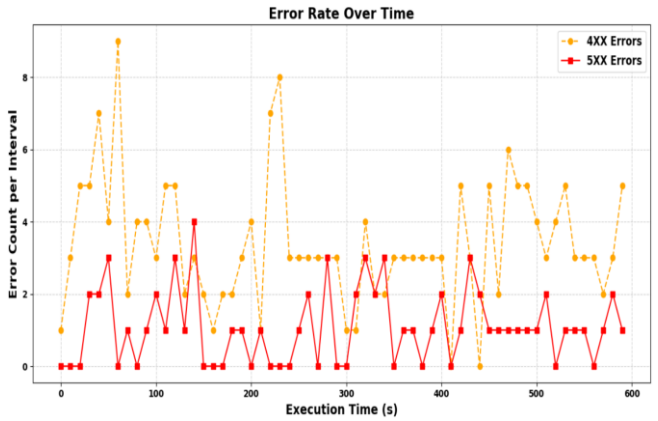

Fig. 8. Error rate over time.

Fig. 8 displays the frequency of 4XX and 5XX errors during a 10-minute microservices operation window. 4XX errors, representing client-side issues, occur more frequently, indicating possible invalid requests or rate limits. 5XX errors are comparatively lower, reflecting stable backend performance. These patterns help evaluate system resilience and pinpoint areas needing request validation or backend hardening.

TABLE IV. KEY PERFORMANCE INDICATORS OF PROPOSED MICROSERVICES ARCHITECTURE

Metric	Baseline	Proposed System	Improvement %
Avg. Latency (ms)	200	80	60%
Throughput @ 4 replicas	2500 req/s	3600 req/s	+44%
Fault Recovery (s)	45	12	-73% faster

Table IV shows the proposed system demonstrates 60% lower latency, 44% higher throughput, and 73% faster fault recovery compared to the baseline, highlighting significant performance, scalability, and resilience improvements in real-time applications.

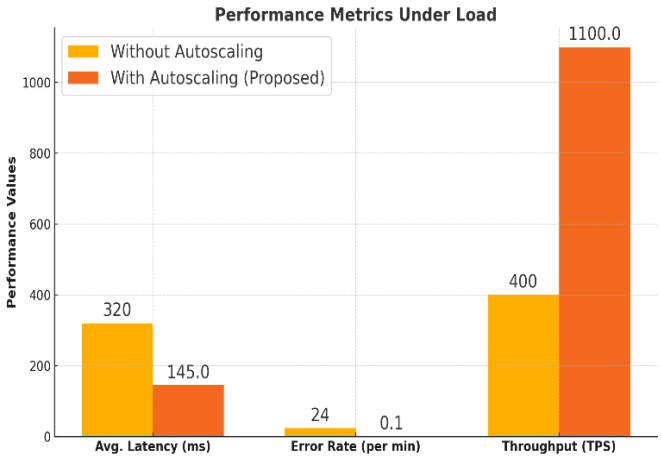

Fig. 9. Comparative performance metrics with and without autoscaling.

Fig. 9 illustrates the impact of autoscaling on system performance under load. Compared to the static architecture, the proposed autoscaling model significantly reduces average latency and error rate while substantially increasing throughput. These improvements validate the efficiency, responsiveness, and scalability of the microservices framework when integrated with intelligent, adaptive autoscaling strategies like KEDA and HPA.

TABLE V. PERFORMANCE COMPARISON OF MICROSERVICE ARCHITECTURES

Model / Research	Avg. Latency (ms)	Throughput (TPS)	Error Rate
Fiber RPC [32]	~50 ms	~300 TPS	Not reported
Pattern-based [33]	~180 ms	~500 TPS	~0.02 %
Fixed-resource benchmarking approach [34]	~150 ms (at 200K TPS)	Up to 1M msg/s	Not specified
Autoscaled (Proposed)	145 ms	1,100 TPS	~0.1 errors/min (~0.0017 %)

Table V shows a comparative performance analysis in terms of latency, throughput, and error rate for three contemporary microservices design patterns and the auto-scaled architecture proposed in this work. The comparison concludes that the proposed architecture not only manages greater concurrency but also ensures noticeably less latency and error rates compared to pattern-based and fiber RPC methods. Additionally, it provides better throughput, which speaks to its increased efficiency and scalability. These enhancements exemplify the architecture's resilience and flexibility in managing real-time workloads, confirming its efficacy in providing fault-tolerant, high-performance service orchestration in dynamic cloud-native deployments with high processing demands.

A. Discussion

Modern cloud-native systems demand architectural designs that can handle high-volume, real-time data with minimal latency, dynamic scalability, and resilience [35]. This research presents a modular microservices-based framework that disaggregates complex system functions into lightweight, independently deployable components. Real-time weather data streams, ingested from APIs like OpenWeatherMap, are processed through a pipeline of containerized services, each managed by Kubernetes for automated deployment, health monitoring, and elastic scaling. Docker containers encapsulate individual microservices such as data ingestion, preprocessing, transformation, and storage—ensuring clear functional boundaries, easier updates, and platform independence. Technologies like Kafka and Flink are selected for high-throughput, low-latency streaming; Redis for fast in-memory caching; Kubernetes with HPA/KEDA for elastic scaling; Docker for isolation and portability; and ELK/Jaeger for observability—together ensuring real-time performance, resilience, and maintainable cloud-native architecture. Adding in-memory caching using Redis and API Gateway as a means of routing and access control reinforce performance along with security.

The Autoscaling tool such as HPA and KEDA automatically scales or scales down the services reduction/increase replicas according to the varying workloads and centralized logging (ELK stack) and distributed tracing (Jaeger) ensure a high level of observability between services. CI / CD pipelines are used to eliminate manual steps that carry out testing, deployment and versioning to guarantee agility and quick iteration. Nevertheless, the system will be forced to overcome issues such as augmented inter-service communications latency, systems complexity of coordination, and distributed data consistency. For instance, high inter-service latency can delay requests, while inconsistent reads across distributed databases may return outdated information, illustrating practical challenges developers face in maintaining system performance and reliability. Additions of service mesh, and edges edge-cloud synchronization, although helpful when relating to resilience and locality, can provide latency and orchestration overhead. Nevertheless, the proposed architecture has been seen to provide a versatile, resilient way of developing cloud-native infrastructures that can power mission-critical, real-time applications in a wide range of domains.

VI. Conclusion and Future Work

The study also offers a scalable and resilient microservices arrangement optimized to the later use of real-time data processing in cloud-native settings. The proposed system is capable of solving the fundamental issues of latency, fault tolerance, and scaling throughout containerization, Kubernetes-orchestration, an event-driven details pipeline, as well as high-quality observability devices. Connection of Kafka, Flink, Redis, and Api gateway allows smooth data processing, transforming, and engaging safe client access with distributed services. The auto scaling will be based on Kubernetes, that along with CI/CD automation, will allow maintenance of response to dynamic workloads in the architecture and continuous integration and deployment. Moreover, end-to-end visibility of the system can be achieved with logging and distributed tracing solutions such as ELK stack and Jaeger that allow proactive diagnostics and optimization of the system. The architecture has proved to be efficient under circumstances where live weather data has been used, and as such, it has proven to be useful across all wider real-time applications, including smart cities, industrial IoT and adaptive analytics. In sum, the given work forms a solid basis to construct smart, componentized and production-ready cloud systems that can address the expansion in need in real-time performance, efficiency and deployment flexibility in the digital ecosystems of today.

The future work now highlights potential bottlenecks, complexity, and scalability challenges, detailing AI-enabled orchestration, edge-cloud federated microservices, multimodal data integration, cost optimization, and dynamic policies to improve real-time responsiveness and large-scale, sustainable deployment. The research will continue to work on an additional basis and research how to incorporate the use of AI enabled services orchestration in order to improve predictive autoscaling as well as anomaly detection. Additional support of edge-cloud federated microservices with the help of the architecture may contribute to the mitigation of the latency and substantially increase the real-time responsiveness of distributed applications across various geographical areas. Besides, involving multimodal data (e.g., images, text) and digital twin diagrams will allow achieving the possibility of the more elaborate level of analytics. Consideration of cost-optimization strategies, the multi-cloud deployment strategies and dynamic policy implementation strategies will also be essential towards enabling larger-scale scalability and sustainability.

References

[1] V. Veeramachaneni, “Edge Computing: Architecture, Applications, and Future Challenges in a Decentralized Era,” Recent Trends in Computer Graphics and Multimedia Technology, vol. 7, no. 1, pp. 8–23, 2025.

[2] S. Gathu and others, “High-Performance Computing and Big Data: Emerging Trends in Advanced Computing Systems for Data-Intensive Applications,” Journal of Advanced Computing Systems, vol. 4, no. 8, pp. 22–35, 2024.

[3] M. Achanta, “The Impact of Real-Time Data Processing on Business Decision-making”.

[4] M. Felisberto, “The trade-offs between Monolithic vs. Distributed Architectures,” arXiv preprint arXiv:2405.03619, 2024.

[5] G. Ortiz et al., “A microservice architecture for real-time IoT data processing: A reusable Web of things approach for smart ports,” Computer Standards & Interfaces, vol. 81, p. 103604, 2022.

[6] D. K. Pandiya and N. Charankar, “Integration of Microservices and AI for Real-Time Data Processing”.

[7] G. Dobriţa, “Adaptive Microservices for Dynamic E-commerce: Enabling Personalized Experiences through Machine Learning and Real-time Adaptation,” Economic Insights-Trends and Challenges, no. 1, pp. 95–103, 2023.

[8] A. Owen, “Microservices Architecture and API Management: A Comprehensive Study of Integration, Scalability, and Best Practices,” 2025.

[9] G. Blinowski, A. Ojdowska, and A. Przybyłek, “Monolithic vs. microservice architecture: A performance and scalability evaluation,” IEEE access, vol. 10, pp. 20357–20374, 2022.

[10] S. Banala, “DevOps Essentials: Key Practices for Continuous Integration and Continuous Delivery,” International Numeric Journal of Machine Learning and Robots, vol. 8, no. 8, pp. 1–14, 2024.

[11] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice api evolution in practice: A study on strategies and challenges," arXiv preprint arXiv:2311.08175, 2023.

[12] S. Ris, J. Araujo, and D. Beserra, "A systemic mapping of methods and tools for performance analysis of data streaming with containerized microservices architecture," in 2023 18th Iberian Conference on Information Systems and Technologies (CISTI), IEEE, 2023, pp. 1–6.

[13] E. Ok and J. Eniola, "Optimizing Performance: Implementing Event-Driven Architecture for Real-Time Data Streaming in Microservices," 2024.

[14] F. Nevola, "Securing communication between microservices in a multi-cloud scenario using Istio service mesh," PhD Thesis, Politecnico di Torino, 2023.

[15] V. Yussupov, "Architectural principles and decision model for Function-as-a-Service," PhD Thesis, Dissertation, Stuttgart, Universität Stuttgart, 2024, 2024.

[16] N. Rathore and A. Rajavat, "Scalable edge computing environment based on the containerized microservices and minikube," International Journal of Software Science and Computational Intelligence (IJSSCI), vol. 14, no. 1, pp. 1–14, 2022.

[17] D. Loconte et al., "Serverless Microservice Architecture for Cloud-Edge Intelligence in Sensor Networks," IEEE Sensors Journal, 2024.

[18] K. Alanezi and S. Mishra, "Utilizing microservices architecture for enhanced service sharing in IoT edge environments," IEEE Access, vol. 10, pp. 90034–90044, 2022.

[19] S. Khriji, Y. Benbelgacem, R. Chéour, D. E. Houssaini, and O. Kanoun, "Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks," The Journal of Supercomputing, vol. 78, no. 3, pp. 3374–3401, 2022.

[20] F. Tusa, S. Clayman, A. Buzachis, and M. Fazio, "Microservices and serverless functions—lifecycle, performance, and resource utilisation of edge based real-time IoT analytics," Future Generation Computer Systems, vol. 155, pp. 204–218, 2024.

[21] O. C. Oyeniran, A. O. Adewusi, A. G. Adeleke, L. A. Akwawa, C. F. Azubuko, and others, "Microservices architecture in cloud-native applications: Design patterns and scalability," International Journal of Advanced Research and Interdisciplinary Scientific Endeavours, vol. 1, no. 2, pp. 92–106, 2024.

[22] S. Ajmal, "Streamlining Fintech Solutions: Cloud-Based Server Management, Scalability Optimization, and Compliance Through Microservices".

[23] S. Semerikov, D. Zubov, A. Kupin, M. Kosei, and V. Holiver, "Models and Technologies for Autoscaling Based on Machine Learning for Microservices Architecture," 2024.

[24] V. S. Kompally, "A microservices-based hybrid cloud-edge architecture for real-time IIoT analytics," Journal of Information Systems Engineering and Management, vol. 10, no. 16s, 2025.

[25] J. A. Rasheedh and S. Saradha, "Design and development of resilient microservices architecture for cloud based applications using hybrid design patterns," Indian J. Comput. Sci. Eng, vol. 13, no. 2, pp. 365–378, 2022.

[26] OpenWeather, "OpenWeatherMap API." 2025. [Online]. Available: https://openweathermap.org/api

[27] E. Serkovas, "Access control approach in microservices architecture," in DAMSS: 15th conference on data analysis methods for software systems, Druskininkai, Lithuania, November 28-30, 2024., Vilniaus universiteto leidykla, 2024, pp. 96–97.

[28] U. Satpathy, H. Borse, and S. Chakraborty, "Towards Generating a Robust, Scalable and Dynamic Provenance Graph for Attack Investigation over Distributed Microservice Architecture," in 2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS), IEEE, 2025, pp. 566–574.

[29] M. Raeiszadeh, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. Mini, "Asynchronous Real-Time Federated Learning for Anomaly Detection in Microservice Cloud Applications," IEEE Transactions on Machine Learning in Communications and Networking, 2025.

[30] M. Zambianco, S. Cretti, and D. Siracusa, "Cost minimization in multi-cloud systems with runtime microservice re-orchestration," in 2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN), IEEE, 2024, pp. 65–72.

[31] P. K. Joshi, "Microservices and API Gateways: The Backbone of Modern Application Platforms".

[32] S. Eyerman and I. Hur, "Efficient Asynchronous RPC Calls for Microservices: DeathStarBench Study," arXiv preprint arXiv:2209.13265, 2022.

[33] W. Meijer, C. Trubiani, and A. Aleti, "Experimental evaluation of architectural software performance design patterns in microservices," Journal of Systems and Software, vol. 218, p. 112183, 2024.

[34] S. Henning and W. Hasselbring, "Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud," Journal of Systems and Software, vol. 208, p. 111879, 2024.

[35] V. Kumar, "Using cloud infrastructure and cloud-native technologies to maximize the scalability and performance of data science applications".