Advance Microservices-Based Approach for Distributed Version Control Processing Using the Sensor-Generated Data by IoT Devices

Amarjeet Singh[1] and Alok Aggarwal[2]

[1]University of Petroleum and Energy Studies, Dehradun, India
ROR
[2]IILM UniversityGr. Noida, India

Correspondence to:
Amarjeet Singh,
amarteotia@gmail.com

Additional material is published online only. To view please visit the journal online.

Cite this as: Singh A and Aggarwal A. Advance Microservices-Based Approach for Distributed Version Control Processing Using the Sensor-Generated Data by IoT Devices. Premier Journal of Science 2025;15:100216

DOI: https://doi.org/10.70389/PJS.100216

Peer Review
Received: 14 August 2025
Last revised: 31 October 2025
Accepted: 17 December 2025
Version accepted: 2
Published: 29 January 2026

Ethical approval: N/a
Consent: N/a
Funding: N/a
Conflicts of interest: N/a

Author contribution:
Amarjeet Singh and Alok Aggarwal–
Conceptualization, Writing – original draft, review and editing

Guarantor: Amarjeet Singh

ABSTRACT

Micro-services, an emerging new architectural style for structuring event-oriented software systems, is gaining attraction among information technology businesses. These applications are typically built as self-contained system, it used concepts of database limited to cater a particular service needs. These services are designed as distinct business functions and deployed utilizing modern tools. Modern features like automatic deployment and CI/CD are commonly implemented in public/private cloud using tried and true DevOps techniques. Every day, a massive amount of data is being generated from many sources, including sensors, website usage, system events, and so on. This fresh flow of events must be handled by modern programs. We can get vital information from these occurrences using event micro-services. In this work, advance micro-services based approach for distributed version control processing using the sensor-generated data by IoT devices has been carried out. The fundamentals of event architecture are examined in this work. The result emphasizes that network capacity remains the first bottleneck in distributed version control for IoT data. Further it is also observed that load balancing and horizontal scaling of repository services are crucial to maintain responsiveness when IoT fleets expand. Minimizing vulnerability windows directly reduces the likelihood of conflicts and improves throughput. It is also observed that the rare but expensive conflicts drastically influence total processing time and reducing retries through conflict-aware scheduling or local buffering leads to significant system-level gains.

Keywords: Apache kafka stream management, CD, Distributed version control integration, Event-driven microservices, IoT sensor event processing, Kubernetes-based CI

Introduction

Event-driven micro-services are nowadays very popular. Event driven micro-services involves defining events as a way to talk with other services within the system.[1–3] It leverages a message broker and follow publish/subscriber pattern. A trade inside reputation of a vital enterprise gadget is defined as an event. A person, as an example, purchases products, registers for a flight, or the bus comes past due somewhere. Irrespective of the industry, events occur everywhere and all of the time. They may be observed in any kind of enterprise. Any activity which is generated, announced, discovered, or utilized as an event is a part of this.

Whilst the event is an occasion, however whenever a message sends the occasion, the two get separated. Event driven micro-services architecture is shown in Figure 1. In this work, advance micro-services based approach for distributed version control processing using the sensor-generated data by IoT devices has been carried out. The fundamentals of event architecture are examined in this work. The established Micro services pattern could be:

- **API Gateway/BFF**: Simplifies client interaction and authentication.
- **Service Meshes (Istio, Linkerd)**: Offer traffic management, mTLS, and observability.
- **Resiliency Patterns**: Circuit breakers, bulkheads, retries, and rate limiting reduce failure propagation.
- **Deployment Strategies**: Blue/Green, Canary, and Rolling updates enable safe delivery.

Events Driven Architecture Using IoT

An event-oriented structure is a system made up of loosely connected micro-services that communicate to one another with aid of creating and consuming events.[4–7] An interest-oriented machine allows messages to be protected in a pastime-orientated environment and in the end published on it.[8–12] IoT events enables the organizations or individuals to perform monitoring of the running equipment or device fleets for failures occurred during the running state. Further, these events also trigger actions whenever such an event occurs. A continuous monitoring of processes, application, related other services and IoT data from various devices is being done by an IoT event.

Based on this continuous monitoring related events are identified and appropriate action is taken. IoT events can be used to build complex event monitoring applications in the Cloud environment. If so, then either IoT events console or REST APIs can be used for review purpose.[13–14] Figure 1 shows the Event driven Micro-services architecture and components in IoT architecture are shown in Figure 2.

Design Fundamentals For Messaging Framework

Apache Kafka is leading in the maximum used streaming data frameworks.[15] Kafka is an established and reliable era that is applied in an extensive range of projects.[16] It can be taken into consideration a geared up-made answer for industrial energy waft processing. Kafka has a massive person base, an active community, and a complete collection of tools. To achieve state repetition, an event replay need to be triggered again in the system. These replayed events then are to be handed over to the event handlers. For applying business logic, now event handler will process these for building the current state of an application. This gives us a way to retrieve any problems that occurred in the past. Precautions have to be taken about event replays. The

Provenance and peer-review:
Unsolicited and externally peer-reviewed
Data availability statement:
N/a

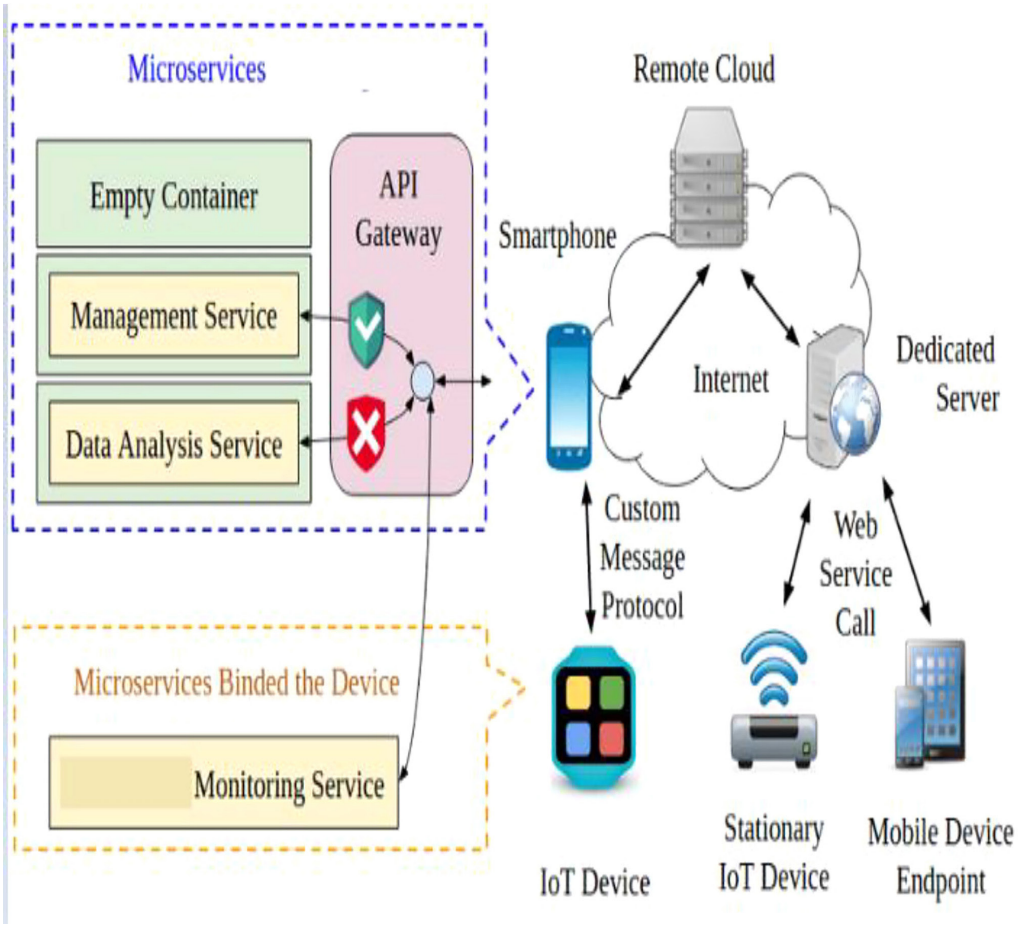

Fig 1 | Event driven Micro-services architecture

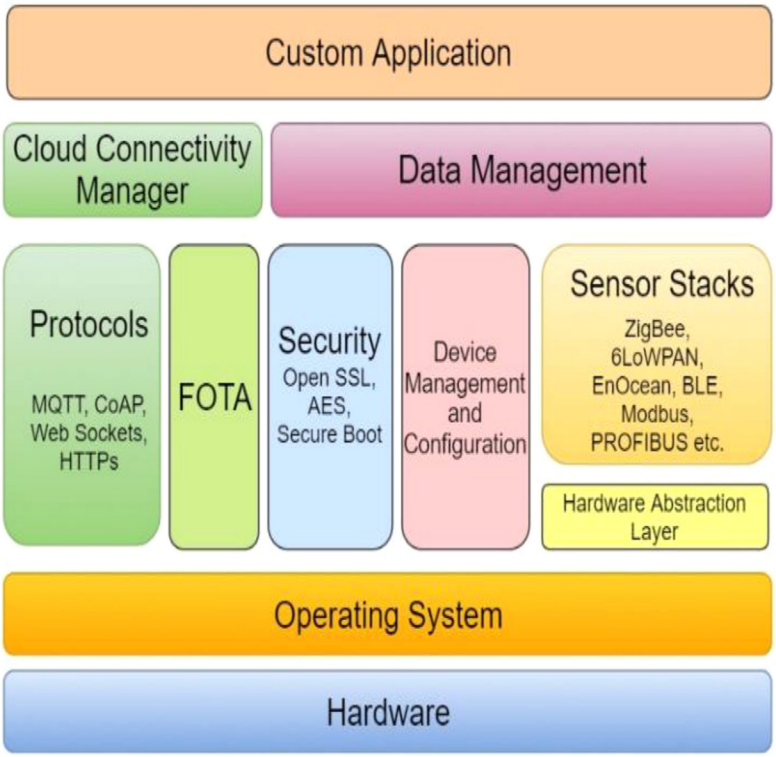

Fig 2 | Components in IoT Architecture

DOI: https://doi.org/10.70389/PJS.100216 | Premier Journal of Science 2025;15:100216

event handler connects with external system. In case of event replays, gateways to other systems has to be disabled. Components interaction in event stream is shown in Figure 3.

With the introduction of CI/CD and APIs for automated infrastructure provisioning, deployment and release methods have developed in recent years. Furthermore, the platforms-as-a-service (PaaS) model's utilization of containers and managed cloud environments necessitates teams working with structures supplied by these technologies/services. Figure 4 depicts the Kubernetes deployment architecture on Amazon Cloud.

Microservices architecture is indeed a transformative approach for developing cloud-native applications, offering numerous advantages such as scalability, resilience, and maintainability. However, organizations often face challenges in adopting microservices effectively, which can lead to sub-optimal implementations. These pitfalls may include:

Complexity in Management: As microservices grow in number, managing them can become increasingly complex. This can lead to difficulties in maintaining service dependencies and orchestrating deployments.

Data Consistency: With microservices, each service typically manages its own database, which can create challenges in ensuring data consistency across services, especially during transactions that span multiple services.

Increased Latency: Communication between services can introduce added latency, impacting application performance—especially when a large number of services are involved in a single user request.

Testing Challenges: Testing microservices can be more complex than traditional monolithic applications due to the need for integration testing and the requirement for a more sophisticated testing strategy.

Cultural Shift: Transitioning to microservices often necessitates a cultural shift within the organization.

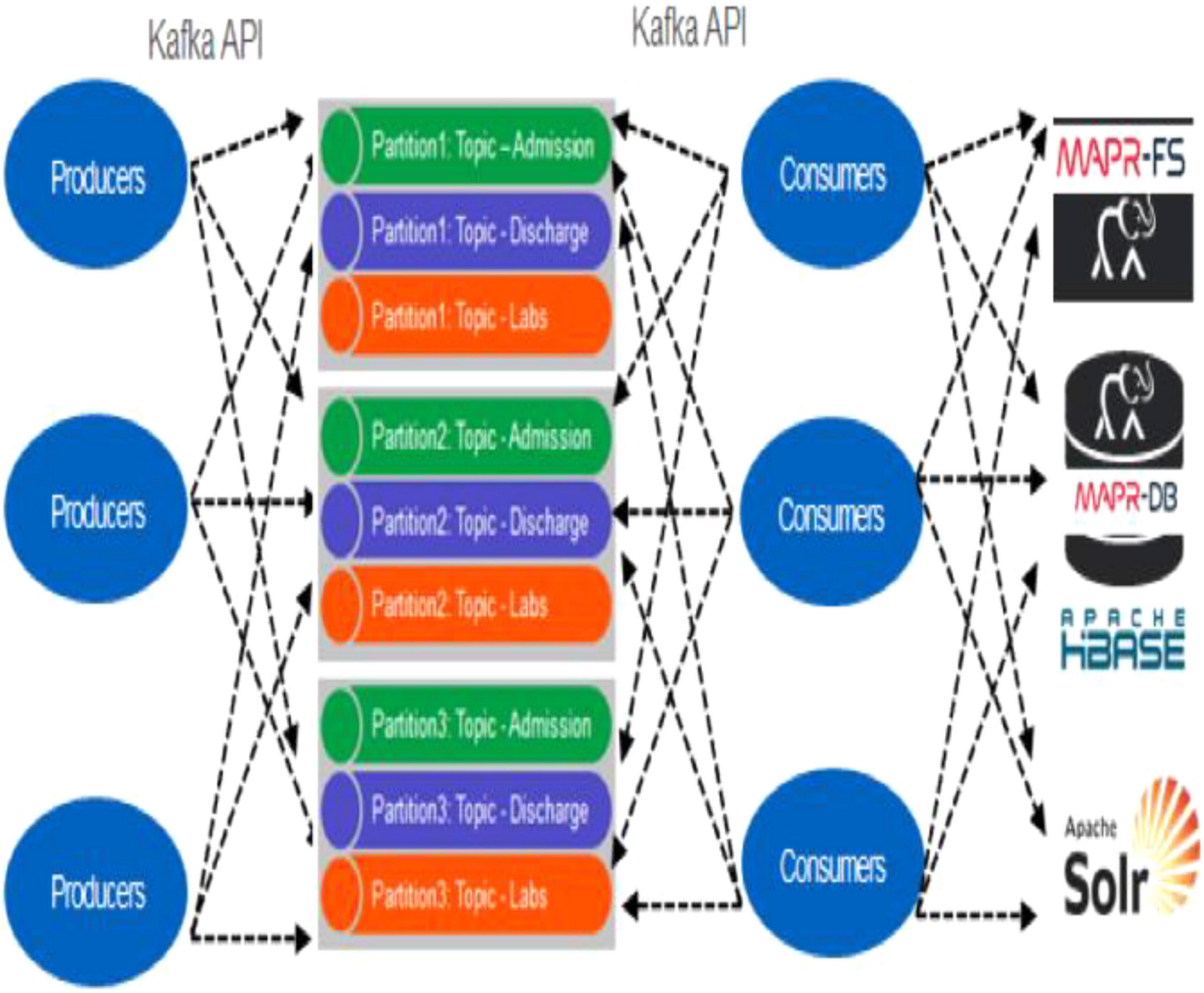

Fig 3 | Components interaction in event stream

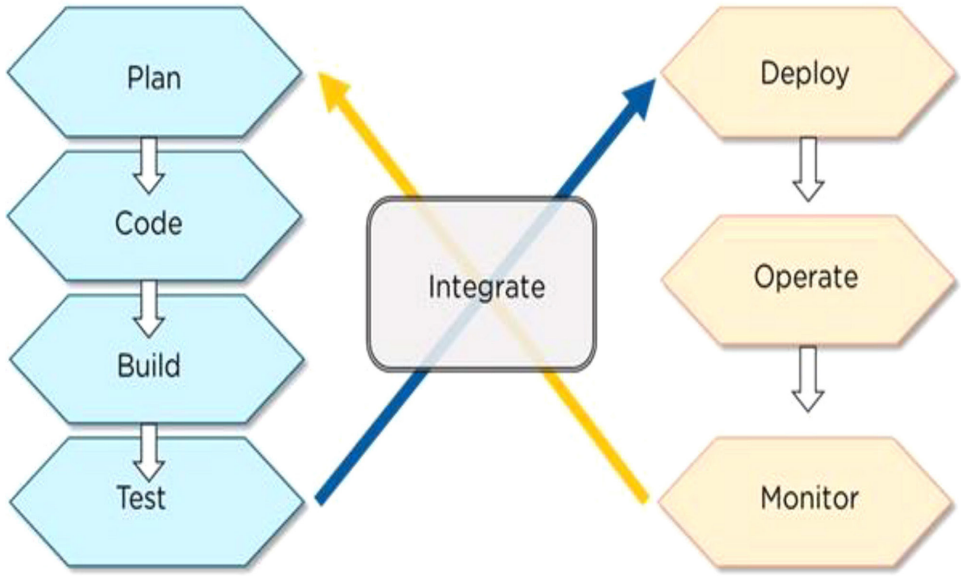

Fig 4 | DevOps methodology in Agile system

Teams must adapt to new roles and responsibilities, and there can be resistance to change from traditional development practices.

To navigate these challenges and reap the benefits of microservices, organizations must implement robust governance and utilize appropriate tools for automation and monitoring. This is where DevOps practices come into play. DevOps complements the microservices approach by fostering collaboration between development and operations teams, leading to:

Enhanced Communication and Collaboration: DevOps emphasizes breaking down silos between teams, which is essential for maintaining the agility that microservices offer. Increased communication leads to faster problem-solving and innovation.

Faster Delivery Times: By integrating continuous integration and continuous deployment (CI/CD) practices, such as those facilitated by Jenkins and similar tools, organizations can accelerate their software delivery processes. This results in more frequent releases and quicker feedback cycles.

Improved Quality and Stability: Continuous testing and deployment practices help catch bugs early in the development cycle, leading to higher-quality software and more stable releases.

Automated Workflows: DevOps tools can automate various processes in the software development lifecycle, reducing the potential for human error and freeing up team members to focus on higher-value tasks.

Increased Productivity and Morale: A culture centered around collaboration and shared responsibility can enhance employee morale and productivity, making teams more effective and engaged in their work.

Scalability: Building CI/CD pipelines allows businesses to scale their operations more effectively, accommodating growth without significant delays or increased risk.

While microservices offer significant benefits, their effectiveness can be maximized through the adoption of DevOps practices and tools. By addressing the inherent challenges of microservices and promoting a collaborative culture, organizations can leverage both microservices and DevOps to achieve their software development goals more effectively. A few of them are listed below.

Delivery of artifacts is faster: DevOps' major goals–automation and continuous delivery – aim to make software development faster and more efficient. DevOps employs numerous strategies to provide a more error-free SDLC flow while employing the Agile method. DevOps embraces a collaborative culture with constant feedback so that any errors are caught early and releases are completed faster. Figure 5 shows a python code for Decision Engine Scaling & Rollback Control.

High cooperation and communication: In the SDLC Ecosystem, development teams must break down all dependencies and antics with proper silo separation and collaborate and communicate. DevOps provides the much–needed atmosphere of mutual cooperation, communication, and integration between globally–distributed teams in an IT organization, paving the path for improved business agility. The stated boundaries are always based on clearly defined duties, which are becoming increasingly important as the DevOps environment grows. It is always available. The quality and timeliness of deliverables are the responsibility of the entire team.

Customer-centric approach: Businesses strive to increase the frequency of their deployments and reduce change failure rates. It is feasible to assure the dependability and stability of an application after each new release by focusing on Devops principles.

Continuous Delivery and Continuous Deployment: Software development processes in the SDLC and Agile

DOI: https://doi.org/10.70389/PJS.100216 | Premier Journal of Science 2025;15:100216

```python
import networkx as nx

def build_dependency_graph(traces):
    """
    traces: list of (caller, callee, latency_ms)
    returns: directed weighted graph
    """
    G = nx.DiGraph()
    for caller, callee, latency in traces:
        if not G.has_edge(caller, callee):
            G.add_edge(caller, callee, calls=0, total_latency=0.0)
        G[caller][callee]["calls"] += 1
        G[caller][callee]["total_latency"] += latency
    # add avg latency
    for u, v, data in G.edges(data=True):
        data["avg_latency"] = data["total_latency"] / data["calls"]
    return G
```

Fig 5 | A python code for decision engine scaling & rollback control

approaches need teams to continuously produce quality software while focusing on speed to market. As the name implies, DevOps brings together two key components of a company's procedures. The plan brings Operations and Creation together in one place. A single umbrella could be a single team with common goals. The convergence of business developments and business goals allows your business team to handle every stage of the DevOps application lifecycle, including growth, testing, and operations.

We need the best parsing of services, clear interfaces for services, and an easy database for each service because the system comprises of multiple Micro-services. Finally, the ability of software firms to tackle these issues is dependent on the infrastructure's maturity, as well as the competency and consistency of design in the company and IT in general.

Methodology

In this study, we model the end-to-end processing time of distributed version control systems handling IoT-generated data. Our methodology integrates both device-side and server-side perspectives. Each IoT device generates commits at a rate λ_D, with each commit of size $|\Delta|$ bytes. The upload time from device to central repository is computed as the sum of network latency and data transfer duration. On the server side, we model commit ingestion using an M/M/1 queuing framework, where the arrival rate Λ aggregates contributions from all devices, and μ represents the server processing capacity.

We incorporate conflict analysis by estimating the probability that concurrent commits target the same branch or resource within a vulnerability window τ. Conflicts are further analyzed by considering the fraction r of automatically resolvable merges versus the fraction requiring human intervention, with associated expected resolution time β. Additional overhead from repush actions after conflict resolution is also quantified.

Local device processing, server ingestion, CI/CD execution, and conflict-related delays are combined into a single expected end-to-end processing time formula. Numerical examples using representative IoT commit sizes, network bandwidths, server capacities, and conflict probabilities are used to validate the model. Sensitivity analysis is performed to understand how each parameter - bandwidth, arrival rate, conflict probability, and auto-merge success rate - affects overall system performance.

This methodology allows us to identify bottlenecks and evaluate strategies such as branch-aware scheduling, edge aggregation, and conflict-aware commit policies. The approach is both analytical and configurable, enabling direct application to real-world IoT–DevOps environments for performance prediction and optimization.

Results And Discussion

Upload Time (Device → Server)

$$T_{\text{upload}} = \ell + \frac{|\Delta|}{B}$$

Explanation: round-trip latency plus pure transfer time.

Example: $|\Delta| = 50{,}000$ bytes, $B = 50{,}000\ \text{B/s}$, $t_\ell = 0.1\text{s} \rightarrow T_{\text{upload}} = 0.1 + 1 = 1.1\text{s}$.

The upload time formula captures the impact of both network latency and raw data transfer rate. In IoT–DevOps environments, small commits generated by sensor data can be frequent, and their efficiency depends strongly on uplink bandwidth. For example, when the commit size is 50 KB, with bandwidth of 50 KB/s and latency of 100 ms, the upload time is about 1.1 seconds. This indicates that even moderate bandwidth constraints can quickly dominate the delay when hundreds of devices report simultaneously. Hence, optimizing network paths or using edge aggregation significantly reduces overall synchronization delays. The result emphasizes that network capacity remains the first bottleneck in distributed version control for IoT data.

Server Queue Wait Time

$$W_{\text{sever}} = \frac{\rho_{srv}}{\mu(1-\rho_{\text{srv}})},\ \rho_{\text{srv}} = \frac{\Lambda}{\mu}$$

Exaplantion: expected wait at server given arrival rate Λ and service rate μ. use only when $\rho_{srv} < 1$

Example: $\Lambda = 2/\text{s}, \mu = 5/\text{s} \rightarrow \rho = 0.4, W = 0.4/(5.0.6) = 0.133\text{s}.$

The server queue wait time highlights the effect of congestion at the repository host. Using an M/M/1 queuing approximation, the expected waiting time depends on the server's service rate and the arrival rate of commits. In our example, with a service rate of 5 commits/sec and arrival rate of 2 commits/sec, the utilization factor is 0.4, leading to a wait of only 0.13 seconds. This suggests that lightly loaded servers introduce negligible delay. However, as utilization approaches 1, waiting time grows rapidly and destabilizes the system. Thus, load balancing and horizontal scaling of repository services are crucial to maintain responsiveness when IoT fleets expand.

Conflict Probability

$$\text{E}\left[T_{\text{merge}} \mid \text{conflit}\right] = r \cdot T_{\text{auto}} + (1-r)\cdot\beta$$

Explanation: mix of automatic merge time and manul human time. if auto merging not availabe, set $r = 0$.

Example: $r = 0.6, T_{\text{auto}} = 2\text{s}_{\text{l}}\ \beta = 600s \rightarrow \text{E}$
$= 0.6 \cdot 2 + 0.4 \cdot 600 = 240.8\text{s}.$

The conflict probability formula estimates the chance of simultaneous updates colliding within the vulnerability window. For example, with 10% chance of targeting the same resource, a 1 commit/sec arrival rate, and 2-second window, the probability of conflict is roughly 18%. This illustrates that even modest commit rates can cause significant contention when many devices write to the same branch or configuration file. Such conflicts increase DevOps overhead by forcing retries or merges. The result shows that minimizing vulnerability windows, e.g., by faster server ingestion or partitioned resource management, directly reduces the likelihood of conflicts and improves throughput.

Merge/Resolution Time

$$\text{E}\left[T_{\text{merge}} \mid \text{conflit}\right] = \text{r} \cdot T_{\text{auto}} + (1-\text{r})\cdot\beta$$

Explanation: mix of automatic merge time and manul human time. if auto merging not availabe, set $\text{r} = 0$.

Example: $\text{r} = 0.6, T_{\text{auto}} = 2\text{s}_{\text{l}}\beta = 600s \rightarrow \text{E} = 0.6 \cdot 2 + 0.4 \cdot 600$
$= 240.8\text{s}.$

The merge resolution formula quantifies the expected extra delay once a conflict occurs. In our example, with 60% auto-resolution rate and 40% requiring human intervention, the average merge time is ~241 seconds, dominated by manual resolution (10 minutes). This demonstrates that the rare but expensive conflicts drastically influence total processing time. Even a small probability of conflict, combined with high human overhead, can multiply delays. Therefore, DevOps pipelines should emphasize automated merging, conflict prediction, or lightweight rollback mechanisms to reduce dependence on manual intervention.

Repush Overhead

$$T_{\text{repush}} \approx T_{\text{uplode}} + W_{\text{server}} + c_{\text{server}}$$

Explanation: time to re-upload and re-process after resolving conflict(simple sum).

c_{server} = server ingest time (sec).

The repush overhead represents the penalty after resolving a conflict, where a device must re-upload its changes. In our simple case, this adds another ~1.4 seconds per failed attempt (upload + queue + processing). While small in isolation, this overhead compounds with higher conflict rates, creating cascading delays across the pipeline. In IoT scenarios with thousands of commits per hour, even modest repush costs consume bandwidth and server cycles. The result stresses that reducing retries through conflict-aware scheduling or local buffering leads to significant system-level gains.

End-to-End Exp-ected Processing Time

$$\text{E}\left[T_{E2E}\right] = c_{\text{local}} + T_{\text{upload}} + W_{\text{server}} + c_{\text{server}} + T_{\text{CI}} + p_{\text{conflict}}$$
$$\left(\text{E}\left[T_{\text{merge}} \mid \text{confict}\right] + T_{\text{repush}}\right)$$

Explantion: base path plus expected added time conflicts occur.

Quick numeric toy:

- $c_{\text{local}} = 0.2\text{s}, T_{\text{upload}} = 1.1\text{s}, W_{\text{server}} = 0.13\text{s}, c_{\text{server}} = 0.2\text{s}, T_{\text{CI}} = 5\text{s},$
- $p_{\text{conflict}} = 0.18, \text{E}\left[T_{\text{merge}}\right] = 240.8\text{s}, T_{\text{repush}} = 1.4\ \text{s}$

Then $\text{E}\left[T_{\text{E2E}}\right] \approx 0.2 + 1.1 + 0.13 + 0.2 + 5 + 0.18(240.8 + 1.4)$
$\approx 6.63 + 44.2 \approx 50.8\text{s}.$

The final end-to-end formula combines all elements: local processing, upload, server wait, CI/CD execution, and expected penalties from conflicts. Using example parameters, the base path without conflicts takes ~6.6 seconds. However, with 18% conflict probability and costly merges, the overall expected time inflates to ~50.8 seconds. This stark increase confirms that conflict handling dominates performance in distributed DevOps for IoT systems. The discussion suggests that while optimizing bandwidth and server load helps, the most critical improvements come from reducing conflict likelihood (ϕ) and improving automatic merge ratio (r). The analysis thus identifies conflict management as the primary research challenge for scalable IoT–DevOps integration.

Conclusion

Microservices architecture offers a robust solution for developing distributed behavioral systems. By breaking down an application into modular, independently deployable services, it enhances flexibility, scalability, and maintainability throughout the software development life cycle (SDLC). Here's a deeper dive into the key aspects and benefits of this architectural approach: Modularity, Independence,

DOI: https://doi.org/10.70389/PJS.100216 | Premier Journal of Science 2025;15:100216

Inter-Service Communication, Technology Agnostic, Resilience and Fault Isolation, Scaling. Microservices architecture offers many benefits like, Faster Time to Market, Ease of Maintenance, Enhanced Collaboration, Improved Resource Allocation, Continuous Delivery and DevOps Alignment. However also suffers from a few challenges like Complexity, Data Management, Deployment Overhead, Inter-Service Latency, Testing Challenges.

The analytical evaluation of distributed version control in IoT–DevOps settings demonstrates that while network bandwidth and server queuing contribute to processing delays, their overall impact remains relatively small compared to the cost of conflict resolution. Upload time is sensitive to device connectivity, and server queues only become critical under high utilization. However, even a modest probability of conflicts, combined with lengthy manual merge resolution, significantly inflates end-to-end processing time. The results highlight that scaling IoT-driven DevOps pipelines is less about raw infrastructure performance and more about minimizing vulnerability windows, reducing the likelihood of overlapping commits, and improving automated merge success rates. In practice, this means designing lightweight synchronization strategies, branch-aware commit scheduling, and intelligent conflict prediction tools. Therefore, the key conclusion is that conflict-aware DevOps pipelines offer the greatest opportunity for performance improvement in distributed IoT version control, ensuring faster, more reliable integration of continuous sensor-generated data.

References

1 Singh V, Singh A, Aggarwal A, Aggarwal S. A digital transformation approach for event driven micro-services architecture residing within advanced VCS. CENTCON. 2021:100–5.

2 Singh V, Singh A, Aggarwal A, Aggarwal S. DevOps based migration aspects from legacy version control system to advanced distributed VCS for deploying micro-services. Proc Int Conf CSITSS. 2021:1–5.

3 Singh V, Alshehri M, Aggarwal A, Alfarraj O, Sharma P, Pardasani KR. A holistic, proactive and novel approach for pre, during and post migration validation from subversion to git. Comput Mater Continua. 2021;66(3):2359–71.

4 Kumar A, Aggarwal A. Lightweight cryptographic primitives for mobile ad hoc networks. RTCNDSSCCIS. 2012;335:240–51.

5 Kumar A, Aggarwal A, Charu. Performance analysis of MANET using elliptic curve cryptosystem. ICACT. 2012:201–6.

6 Goyal M, Aggarwal A. Composing signatures for misuse intrusion detection system using genetic algorithm in an offline environment. AISC. 2012;176:151–7.

7 Vishnoi A, Aggarwal A, Prasad A, Prateek M, Aggarwal S. A cryptosystem analysis for text messages using homomorphic transform. ICICICT. 2022:1445–50.

8 Kumar A, Gopal K, Aggarwal A. A complete, efficient and lightweight cryptography solution for resource constraints mobile ad-hoc networks. PDGC. 2012:854–60.

9 Mittal S, Aggarwal A, Maskara SL. Application of Bayesian belief network for context extraction from wireless sensors data. ICACT. 2012:410–5.

10 Vishnoi A, Aggarwal A, Prasad A, Prateek M, Aggarwal S. An improved cryptographic technique using homomorphic transform. ICICICT. 2022:1451–4.

11 Singh T, Srivastava DK, Aggarwal A. A novel approach for CPU utilization on a multicore paradigm using parallel quicksort. CCICT. 2017:1–6.

12 Kumar A, Gopal K, Aggarwal A. Simulation and cost analysis of group authentication protocols. IC3. 2016:1–7.

13 Madam C, Aggarwal A, Cheng X, Rani A, Kumar M, Shankar A. A non-invasive approach to identify insulin resistance with triglycerides and HDL-c ratio using machine learning. NPL. 2021;52(3).

14 Aggarwal S, Tomar SK, Aggarwal A. Performance analysis of soft handoff algorithm using fuzzy logic in CDMA systems. PDGC. 2012:586–91.

15 Vishnoi A, Aggarwal A, Prasad A, Prateek M, Aggarwal S. Image encryption using homomorphic transform. ICICICT. 2022:1455–9.

16 Gupta SK, Govil K, Agarwal A. Routing algorithm for energy conservation in MANET. CICN. 2015:165–7.