The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 157-165
DOI 10.37547/tajet/Volume07Issue04-21

Check for updates

Implementing Service Mesh Architecture for Scalable Applications

Mantu Singh
Software Architect, Reged
Acton, USA

OPEN ACCESS

SUBMITTED 17 February 2025
ACCEPTED 25 March 2025
PUBLISHED 30 April 2025
VOLUME Vol.07 Issue 04 2025

CITATION

Mantu Singh. (2025). Implementing Service Mesh Architecture for Scalable Applications. The American Journal of Engineering and Technology, 7(04), 157–165. https://doi.org/10.37547/tajet/Volume07Issue04-21

Abstract: This study examines a decentralized approach to implementing a service mesh for microservice-based systems designed for scalable data processing. Unlike traditional solutions dominated by the pipes-and-filters pattern and a centralized control plane, this approach utilizes the concept of Eblocks—unified modules that incorporate service discovery, authentication, monitoring, and load management components. This allows for the formation of various patterns (manager-worker, divide-and-conquer, hybrid models) directly at the microservice level without strict dependence on centralized logic. It is demonstrated that such an architecture accelerates data processing through automatic scaling and parallel execution, simplifies configuration, and provides flexible security and observability mechanisms. The proposed results, supported by findings from other researchers, indicate a significant increase in system throughput when handling documents requiring pipeline, parallel, and distributed processing. The presented information is of interest to researchers and professionals in distributed systems, cloud computing, and microservice architecture, aiming for a deeper understanding and implementation of innovative service mesh architectures to enhance the scalability, reliability, and efficiency of modern IT applications.

Keywords: cloud computing, microservices, service mesh, processing patterns, scalability, decentralized architecture, Eblocks.

Introduction: Cloud computing provides flexible and dynamic access to computational resources with minimal management costs. The shift from monolithic applications to microservices enhances autonomy, elasticity, and accelerates the deployment cycle of new features. Microservice architecture promotes

modularity and simplifies the maintenance of large-scale systems; however, it presents engineers with several challenges, including the complexity of managing interactions among independent services, ensuring observability, securing communication, and enabling automatic scaling.

To address these challenges, service mesh platforms have emerged, handling traffic routing, load balancing, encryption, monitoring, and service management without modifying business logic. However, most modern service meshes are primarily designed around the pipeline processing pattern and lack built-in support for a broader range of processing patterns, such as manager-worker or divide-and-conquer. This limitation restricts developers and architects in designing complex, high-performance systems that require flexible management of parallel data streams and distributed task execution.

A comprehensive review of contemporary research follows. Nicolas-Plata A., Gonzalez-Compean J. L., and Sosa-Sosa V. J. [1], along with Alboqmi R. and Gamble R. F. [4], propose an innovative approach based on service mesh integration to support processing patterns within microservice applications. Parallel to this, the evolutionary transformation of software systems from monolithic to microservice architectures is explored in the works of Akerele J. I. et al. [3], Newman S. [9], and Decimavilla-Alarcón D. C. and Marcillo-Franco P. F. [8]. These studies focus on enhancing scalability and flexibility through cloud technologies and containerization, allowing systems to adapt to specific requirements, such as healthcare or IoT environments.

Another research direction involves developing tools for evaluating scalability and selecting optimal design patterns. Wrona Z. et al. [5] expand the capabilities of a cloud infrastructure simulator to model the dynamics of scalable systems, while Dhait S. et al. [6] conduct a comparative analysis of creational patterns in software development.

Cybersecurity in cloud platforms is also a significant topic in modern literature, as examined in the study by Molnar V. and Sabodashko D. [2]. Their work aims to compare security standards based on NIST guidelines to identify vulnerabilities in leading cloud platforms.

Additionally, scalability is explored in the context of serverless computing trends in the study by Li Y. et al. [7]. Their research summarizes the current state of the technology, identifies key challenges, and outlines future prospects for serverless approaches in cloud services.

A notable gap in the existing research is the absence of a unified mechanism for combining multiple processing patterns, such as the sequential integration of manager-worker and divide-and-conquer models, within a service mesh. Creating such hybrid patterns still requires either manually configuring multiple tools or abandoning the advantages of a transparent service mesh in favor of low-level orchestration. This lack of a universal method for integrating diverse microservice processing patterns into a service mesh model defines the research gap that this study aims to address.

The objective of this study is to examine the implementation features of service mesh architecture for scalable applications.

The scientific novelty lies in the systematic analysis and comparative evaluation of existing research on service mesh architectures for scalable applications, enabling the identification of problem areas and future research directions.

The proposed hypothesis suggests that implementing a comprehensive service mesh, where each microservice is equipped with built-in discovery, authentication, and monitoring functions, will:

- Reduce deployment and configuration time for distributed processing patterns.
- Significantly increase architectural flexibility by supporting additional processing patterns.
- Decrease reliance on traditional proxy-based approaches and centralized controllers while maintaining performance.

The methodological framework of this study includes an analysis of existing research on microservice architecture and service meshes.

1. Theoretical and technological foundations of service mesh

Service mesh initially emerged as a methodology for abstracting network functions and transferring them to a separate layer, independent of the business logic of microservices. Traditionally, service mesh architecture consists of two planes:

- Data Plane: Comprises proxies (sidecars) that attach to each service. These sidecar proxies intercept traffic, route requests between services, and provide authentication, encryption, and tracing mechanisms.
- Control Plane: Manages the configuration of sidecar proxies, monitors network state, balances load, and enforces security policies, such as mandatory traffic encryption and authorization. In some implementations, such as Istio, the control plane operates as a centralized component where microservices register to receive routing policies [1].

Below, Figure 1 illustrates the functions of a service mesh.

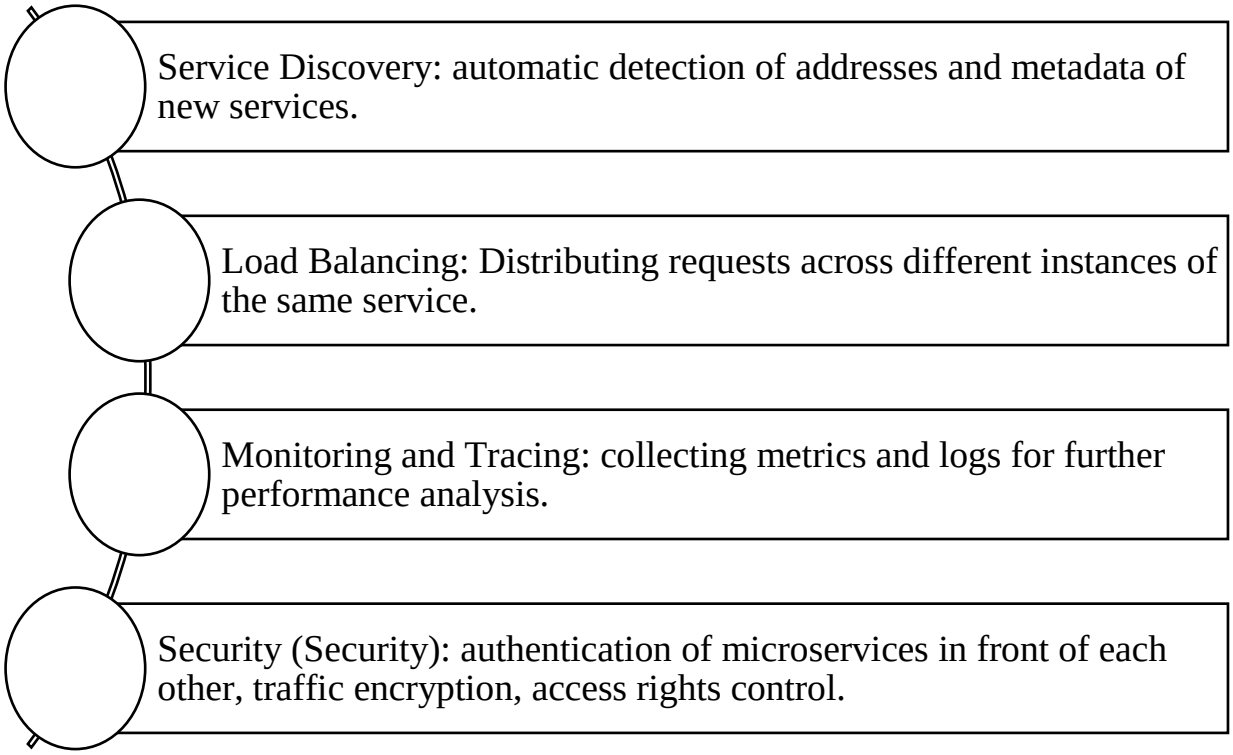

Fig.1. Functions of the service mesh [1].

Despite the clear advantages of service meshes, their implementation requires consideration of several factors, including compatibility with cloud orchestrators (e.g., Kubernetes), configuration complexity, and the resource-intensive nature of proxies running alongside each microservice.

The primary limitations of existing implementations stem from the fact that widely used service mesh platforms predominantly support the Pipes & Filters pattern, as it naturally aligns with the concept of sequential data flow through sidecar proxies. However, more complex patterns such as Manager–Worker or Divide & Conquer remain poorly automated in standard service meshes [7].

The root cause of this limitation lies in the fact that distributed load management (for instance, launching multiple instances of worker microservices) is traditionally delegated to external container orchestration systems such as Kubernetes or Docker Swarm. A service mesh primarily detects new services and, at best, routes traffic to them in a round-robin manner [3]. However, it often lacks a flexible mechanism for defining roles, distributing tasks, and automatically shutting down redundant worker services when load conditions change.

Additional compatibility issues arise in practice when certain microservices require specific network policies, such as gRPC streaming or WebSocket connections, which may not be fully supported by the service mesh. Another challenge is the increased overhead associated with proxy operation, particularly in environments with a high density of microservices [7].

As a result, developers frequently face a dilemma: either limit themselves to the convenience of simple patterns and built-in service mesh mechanisms or manually implement complex scenarios such as Manager–Worker or parallel data partitioning, bypassing standard functionality. This issue underscores the need for new decentralized service mesh solutions that offer greater flexibility in integrating various microservice processing patterns [2, 6].

Table 1 below presents a comparative analysis of microservice processing patterns.

Table 1. Comparative characteristics of microservice processing patterns [1, 5, 9]

Pattern	Key features	Typical applications	Support in most service meshes
Pipes & filters	Sequential data transmission between "filters"; facilitates pipeline processing	Simple ETL processes, log analysis, data transformation	Full (native to service meshes)
Manager–worker	A central "manager" distributes tasks to a pool of "workers"; scales by increasing the number of workers	High-load processes requiring distributed parallel processing	Partial (mainly through orchestrator functionality)
Divide & conquer	Task is divided into smaller parts, processed in parallel, and results are aggregated	Large-scale computational tasks (ML models, big data analysis)	Weak (usually requires manual configuration and coding)

As shown in Table 1, service meshes fully support the classic pipeline-based approach (Pipes & Filters). However, Manager–Worker and Divide & Conquer patterns remain on the periphery of adoption and often require extensive customization at the Kubernetes manifest level and the integration of additional tools.

2. Proposed architectural approach for integrating multiple patterns

This section explores a decentralized service mesh model designed to support flexible data processing scenarios that require the simultaneous use of multiple patterns, such as pipes and filters, manager–worker, and divide and conquer. The primary concept is based on Eblocks, which encapsulate the business logic of a microservice along with key service mesh mechanisms, including authentication, monitoring, service discovery, and, when necessary, automated load management [4, 9].

Traditional service meshes, such as Istio and Linkerd, rely on a control plane and a data plane. However, this architecture centralizes most routing logic, making it difficult to dynamically expand functionality, such as quickly adding new processing patterns. In contrast, the Eblocks-based approach employs the following principles:

1. Decentralized storage of service logic. Each Eblock consists of:

- Processing Microservice (PM): The core business application or function, such as an encryption service or risk analysis module.
- Workload Manager (WM): A local (within the Eblock) implementation of load distribution algorithms that activates predefined roles, including manager, worker, divider, conquer, and combine.
- Discovery: A self-registration and peer-to-peer lookup mechanism, implemented using a distributed hash table or similar technology.
- Authentication: Built-in authentication

mechanisms for issuing and verifying security tokens used for inter-Eblock communication.

- Monitoring: A local metrics collection service accessible via REST API or client libraries [7, 8].

2. Intercomponent communication interfaces. These are based on REST, gRPC, WebSocket protocols, and message buses to ensure reliable routing.

3. Security and authentication mechanisms. These include in-transit and at-rest data encryption, digital certificate management, authentication, and tokenization.

4. Anomaly detection and dynamic risk assessment. Implementation of intrusion detection systems, monitoring algorithms, and dynamic threat evaluation for real-time cybersecurity responses.

5. Adaptive load balancing and scaling. Local and global resource allocation algorithms that analyze current workloads and forecast peak demands.

To implement a specific pattern, groups of Eblocks can assume various roles, such as Filter, Manager, Worker, Divide, and Conquer. For example, in a manager–worker pattern, one Eblock acts as the Manager, handling task segmentation and scheduling, while multiple Eblocks in the Worker role process the workload. Figure 2 illustrates how the Eblocks approach integrates multiple patterns.

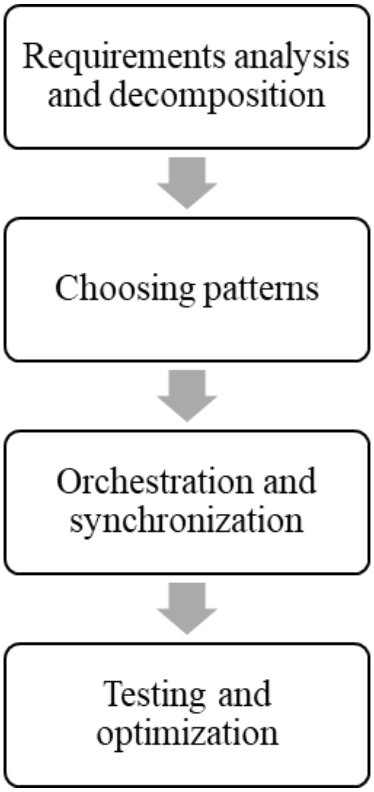

Fig. 2. The scheme of the Eblocks approach in the process of integrating several patterns [1, 4].

For the Eblocks model to function effectively, a comprehensive orchestration strategy is required. This typically relies on Kubernetes or a similar orchestrator but does not duplicate its functions. To provide a clearer understanding, Table 2 compares key elements of the traditional service mesh model (using Istio as an example) and the proposed Eblocks architecture.

Table 2. Comparison of the classical service mesh model and Eblocks [1, 4]

Characteristic	Classical service mesh (e.g., Istio)	Eblocks
Interaction management	Centralized control plane: Istio Control Plane (Pilot, Mixer, Citadel, etc.)	Decentralized model: Each Eblock contains Discovery, Auth, and Monitoring components
Pattern implementation	Default support for pipes and filters; other patterns require complex configuration or external logic	Support for multiple roles (manager, worker, filter, etc.) with local task distribution
Load balancing mechanism	Sidecar proxy (Envoy) surrounds the microservice, handling routing and load balancing	Workload Manager inside each Eblock with pseudo-random or custom algorithms
Monitoring and tracing	Centralized Mixer (pre-Istio v2) or Telemetry input in later versions; requires consistent configuration	Local Monitoring in Eblock with REST API, ensuring uniform metrics within the container
Security and authentication	mTLS between proxies, certification through Citadel, granular RBAC policy	Built-in Auth component for distributed token verification, eliminating unnecessary nodes
Extensibility and customization	High, but requires deep knowledge of Istio CRD and Kubernetes integration	Flexible pattern combinations since roles are defined at the Eblock level
Overhead	Additional sidecar proxies, centralized configuration storage, control plane load	Higher resource consumption within each Eblock than a standalone service, but no central bottleneck

The classical service mesh model offers high configurability for pipelines but complicates load management and parallel execution. In contrast, the Eblocks approach inherently supports these patterns through a local Workload Manager, reducing reliance on a centralized service mesh.

A key distinction of Eblocks is the reduced need for constant interaction with a central controller. The combination of service discovery and authentication within each block removes some network constraints but imposes additional requirements on the self-sufficiency of each Eblock. Nevertheless, the proposed model provides broader capabilities for integrating new pattern combinations.

3. Practical implementation and performance evaluation

An analysis was conducted on the methodology for validating the functionality and efficiency of the approach based on decentralized Eblock components, as proposed by Nicolas-Plata A., Gonzalez-Compean J. L., and Sosa-Sosa V. J. [1]. This method enables the transformation of traditional application sets into modern microservice-based systems by integrating hybrid data processing patterns, optimizing pipeline analysis, encryption, and result storage. The core idea is to create a system capable of dynamically combining the manager–worker and pipeline (pipes and filters) patterns while also incorporating the divide-and-conquer strategy for processing large-scale data workloads.

The primary objective of the system is to ensure efficient pipeline processing and detailed analysis of large text document datasets, followed by mandatory encryption and storage in a centralized repository. To achieve this, a set of specialized microservices was developed, each performing specific functions within the overall architecture. The system includes a semantic analysis module for extracting keywords and thematic attributes from documents, a risk assessment component for evaluating data sensitivity levels, symmetric encryption and decryption services to ensure data security, a database for storing metadata and processing results, a user interface for end-user access, and a routing and request parameterization service, which, in the experimental phase, operates as an independent microservice despite its potential integration into a unified Eblock.

Key system requirements include the ability to create multiple instances of worker modules for parallel document processing, minimizing manual configuration through a declarative architectural approach, and ensuring transparency in monitoring and security processes via built-in service authorization. A comparative analysis of approaches before and after integrating decentralized Eblock components demonstrates significant improvements. The traditional method, where microservices were manually interconnected via REST interfaces and required additional scaling scripts, was outperformed by the new approach, in which each component is packaged into an Eblock with preconfigured Discovery, Authentication, Monitoring, and Workload Manager modules. This allows for seamless activation of manager–worker roles or the implementation of sequential pipeline schemes at the configuration level.

To validate the developed system, a modern hardware and software infrastructure was utilized. The containerization environment was based on Kubernetes version 1.23 (or later), deployed on two worker nodes and a single master node. The server hardware included Intel Xeon processors with 12–16 logical cores and RAM ranging from 64 GB to 128 GB, providing the required computational capacity. The network environment was structured within an internal 10 Gbps network, and user interface access was secured through port 443 (HTTPS). Data storage relied on the distributed SkyCDS system, integrated with a local file system for handling temporary data, ensuring high reliability and fast data access.

The system deployment process consists of multiple sequential stages. First, Docker images were built for each microservice, embedding the necessary Eblock components such as Discovery, Authentication, Monitoring, and, when required, Workload Manager for Manager or Worker roles. The next step involved generating YAML configurations where a DevOps engineer defined functional roles and interactions between system components. A dedicated generator processed these descriptions to create Kubernetes manifests (Deployments, Services, ConfigMaps), specifying container launch parameters and internal component initialization. The final stage involved the step-by-step deployment and initialization of Eblock components while considering dependencies: the Discovery module identified other instances through a distributed hash table, Authentication components exchanged tokens, and Monitoring aggregated metrics for subsequent analysis.

System performance was assessed using a set of metrics characterizing the overall architecture's efficiency and reliability. Key measurements included:

- Response Time (RT): The time interval from sending an HTTP request to receiving a response.
- Pattern Response Time (PRT): The total response time of all microservices involved, including additional coupling time (CT) required for result aggregation.
- Throughput (TPS or docs/sec): The number of documents processed per unit of time.
- Infrastructure overhead: Evaluated in terms of memory consumption, CPU load, and dynamic scaling time for worker nodes.

Load tests were conducted using a dataset of 2,745 text documents, with an average size of approximately 4 KB. Some documents required computationally intensive operations, such as semantic analysis and encryption using long keys dependent on risk level. The experiment involved three data processing scenarios:

1. Linear pipeline processing: Each document sequentially passed through semantic analysis, risk assessment, encryption, and was finally stored in a database.

2. Manager–worker decryption: One Eblock acted as the Manager, dynamically launching between one and four worker nodes for parallel processing, while other components functioned in filtering mode.

3. Hybrid processing: A combination of sequential pipeline processing and dynamic scaling at the decryption stage using a worker Eblock pool.

Table 3 presents a comparison of the average throughput (docs/sec) across different pattern integration scenarios.

Table 3. Throughput (TPS) in different scenarios of pattern integration [1]

Scenario	Number of worker nodes	Average TPS (docs/sec)	Increase from baseline (A)
A (Pipeline)	—	12.4	—
B (Manager–Worker)	2	15.8	+27%
B (Manager–Worker)	4	17.5	+41%
C (Hybrid)	2 (Decipher)	16.3	+31%
C (Hybrid)	4 (Decipher)	18.1	+46%

(Data averaged over multiple test runs, margin of error within ±2%.)

The analysis demonstrated a significant reduction in overall system response time when transitioning from a traditional linear pipeline to a hybrid solution incorporating the manager–worker pattern. On average, response times decreased by 23–27%, primarily due to efficient parallelization of computational processes, particularly in decrypting large files. Additionally, dynamic scaling of worker nodes increased system throughput by approximately 35% compared to the baseline configuration.

The results indicate the strong potential of the proposed approach for building scalable and resilient microservice architectures. The combination of multiple patterns and automated worker node deployment accelerates data processing by up to 46% compared to a linear pipeline without parallelization.

CONCLUSION

The study examined the limitations of existing service meshes, which are primarily designed around the pipeline processing pattern. It has been shown that this approach is not always suitable for scenarios requiring dynamic scaling and the combination of multiple patterns, such as manager–worker and divide-and-conquer. A decentralized service mesh implementation has been proposed, where key tasks such as service discovery, authentication, monitoring, and load distribution are handled by the microservices themselves in the form of Eblock structures.

Experimental results confirm that the described architecture reduces configuration overhead, simplifies the addition of new patterns, and significantly accelerates data processing for large workloads. Furthermore, the system has demonstrated adaptability to varying load conditions through the automatic deployment of additional worker instances. It is important to note that decentralization imposes specific resource requirements but effectively balances the load, eliminating the bottleneck associated with a centralized controller.

Future research directions include the development of tools for the automatic selection of the optimal pattern or their combination based on workload profiles, as well as the expansion of the service mesh with intelligent scaling and load forecasting capabilities. Additionally, further integration of the Eblocks mechanism with standard orchestration systems such as Kubernetes and Docker Swarm is planned, along with an in-depth evaluation of efficiency in real-world industrial use cases.

REFERENCES

Nicolas-Plata A., Gonzalez-Compean J. L., Sosa-Sosa V. J. A service mesh approach to integrate processing patterns into microservices applications //Cluster Computing. – 2024. – Vol. 27 (6). – pp. 7417-7438.

Molnar V., Sabodashko D. Comparative analysis of cybersecurity in leading cloud platforms based on the

NIST framework //Social Development and Security. – 2024. – Vol. 14 (6). – pp. 68-80.

Akerele J. I. et al. Improving healthcare application scalability through microservices architecture in the cloud //International Journal of Scientific Research Updates. – 2024. – Vol. 8 (2). – pp. 100-109.

Alboqmi R., Gamble R. F. Enhancing Microservice Security Through Vulnerability-Driven Trust in the Service Mesh Architecture //Sensors. – 2025. – Vol. 25 (3). – pp. 914.

Wrona Z. et al. Scalability of Extended Green Cloud Simulator //2024 International Conference on INnovations in Intelligent SysTems and Applications (INISTA). – IEEE. - 2024. – pp. 1-6.

Dhait S. et al. Analysis Of The Best Creational Design Patterns In Software Development //2024 8th International Conference on Computing, Communication, Control and Automation (ICCUBEA). – IEEE. - 2024. – pp. 1-5.

Li Y. et al. Serverless computing: state-of-the-art, challenges and opportunities //IEEE Transactions on Services Computing. – 2022. – Vol. 16 (2). – pp. 1522-1539.

Decimavilla-Alarcón D. C., Marcillo-Franco P. F. Arquitectura de microservicios basada en contenedores para despliegue ágil de aplicaciones IoT en la nube //Revista Científica Episteme & Praxis. – 2025. – Vol. 3 (1). – pp. 35-49.

Newman S. Monolith to microservices: evolutionary patterns to transform your monolith. – O'Reilly Media. - 2019. – pp. 5-20.