ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Service Mesh: The Rise of Event-Driven Asynchronous Mesh in Cloud Continuum

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Montebugnoli, S., Sabbioni, A., Foschini, L., Corradi, A. (2026). Service Mesh: The Rise of Event-Driven Asynchronous Mesh in Cloud Continuum. IT PROFESSIONAL, 28(1), 51-58 [10.1109/MITP.2025.3605101].

Availability:
This version is available at: https://hdl.handle.net/11585/1050253 since: 2026-02-26

Published:
DOI: http://doi.org/10.1109/MITP.2025.3605101

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

18 July 2026

Article Type: Description

Service Mesh: the Rise of Event-Driven Asynchronous Mesh in Cloud Continuum

Sofia Montebugnoli, Andrea Sabbioni, Luca Foschini, Antonio Corradi, *DISI - University of Bologna, Bologna, 40136, Italy*

Abstract—The proliferation of microservice-based IT applications has increased the complexity of big data processing within any modern cloud application, requiring new strategies for managing, controlling, and accelerating deployments. Service Mesh offers new management support that facilitates control over a very large number of services while separating logic from control specification. Applications becoming more decentralized and deployed across multiple regions in the cloud continuum ask for even more uncoupled and asynchronous interactions to enhance reliability and scalability. Event-driven architectures offer a viable solution; consequently, Event Mesh emerges as a cloud paradigm promoting event-based, asynchronous interactions among services and data. This paper compares Service Mesh and Event Mesh approaches, claiming that Event Mesh improves uncoupling and offers asynchronous behavior in distributed systems to optimize performance in traffic management, observability, and security.

The pervasiveness and emergence of IT technologies in a variety of different areas, such as Next-Generation Networks, Smart Cities, and Industry 5.0, has produced a widespread and complex Cloud Infrastructure organization to support the ever-increasing need of IT services. In this sense, microservice architectures have started supporting these increasing complexities by enabling greater scalability, flexibility, and agility in software development and deployment [1].

Different established settings, such as private, public, and hybrid clouds, as well as novel distributed deployment environments, including targets like edge, fog, and cloud nodes, further harden the meeting of the above requirements. These new forms of cloud continuum require management, orchestration, and fine-grained communication among services.

A potential solution for communication management in large distributed applications is Service Mesh, that simplifies communication management by providing features for **traffic, observability, and security**. These three features are crucial for managing complex microservices architectures by offering flexibility in configuration and separating control features from the application [2]. In addition, the separation prevents the introduction of ad-hoc, heavy solutions where infrastructure concerns are mixed with the application code [3].

Service Meshes provide an infrastructure layer to support communication among microservices [2], separating the application from the management of infrastructure concerns related to the control and configuration of microservices leveraging the available features of the mesh. This paradigm makes possible the routing of all traffic through a networking layer, relieving developers from implementing infrastructure capabilities and enabling system administrators to control and configure the application.

This paper proposes a comparison of both Service and Event Mesh approaches, since the latter promotes a more asynchronous and decoupled interaction among microservices through Event-driven communication. To manage communication in applications decoupled in both space and time, and also distributed across a variety of heterogeneous resources, an Event Mesh exploits Message-Oriented Middleware (MOM) to implement event-driven communication effectively. For both Service and Event Mesh, we compare essential features for both developers and service administrators managing distributed cloud applications, including control and configuration. Additionally, we present

XXXX-XXX © 2025 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

existing technologies implementing Service and Event Mesh paradigms and their possible evolution. We argue about the central role that the mesh paradigm will play in future IT services due to the separation of infrastructure and application concerns it introduces, fastening application development and promoting good DevOps practices.

Why Modern Cloud Applications Need a Mesh

Modern cloud applications employ microservice architectures to isolate business logic into a self-contained module, so facilitating independent business logic development, deployment, and service scaling [4]. Scaling on-demand the replicas and resources of micro components optimizes resource allocation and enhances cost efficiency. The separation of developed functions into isolated services facilitates the identification of faults and troubleshooting by minimizing the impact of failures on the overall system. With Continuous Integration and Continuous Deployment (CI/CD), each service can be tested, built, and deployed independently, allowing for faster and more frequent releases with reduced risks, enhancing agility and flexibility in the maintenance of services during their lifecycle.

However, the adoption and maintenance of a microservice architecture introduce complexity and fragmentation in the control of the development and the configuration of an enormous number of application components. The microservice model in itself still lacks a unified control solution to support the increasing complexity of large distributed deployments in particular for: i) *configuration of the communication* among services; ii) *traffic management* between microservice [5]; iii) *very different tools due to the lack of widely adopted standards for observability* of application and its instrumentation [6], and iv) *security in service-to-service* communication [2].

In that increasingly widespread context, the Service Mesh approach addresses the challenges that developers face in managing cross-infrastructure and application concerns in modern cloud application deployments.

Service Mesh

Service Mesh works as an **infrastructure layer** integrated into modern cloud platforms to offer developers and service administrators many facilities and abstractions for developing, deploying, and managing microservice-based applications [2]. Service Mesh offers a **data plane** to handle data traffic between microservices via **proxy instances** deployed alongside each service.

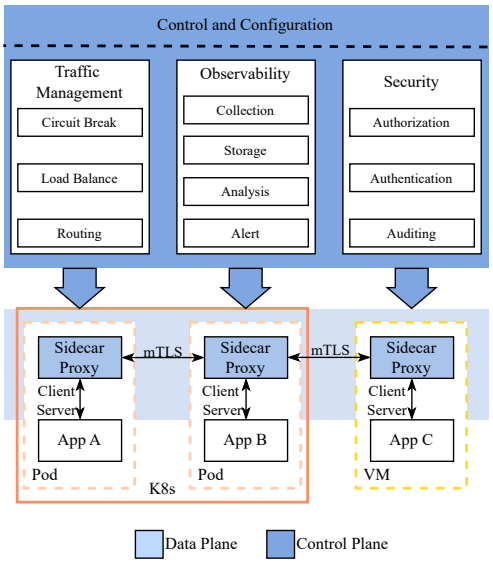

FIGURE 1. High-level abstraction of Service Mesh architecture, showing the communication in the data plane with services, devices, and virtual machines, and of the pivotal functions of the control plane

The **control plane** is able to centrally manage and configure all control policies by providing administrators with centralized control and visibility over the behavior of the Service Mesh. The control plane typically consists of several components, including **control plane APIs, one configuration store**, and **many controllers** available to manage and better adapt to system conditions dynamically.

Fig. 1 shows the architecture of Service Mesh usually implemented by following the **Sidecar pattern** [2]. The Sidecar Proxy is deployed along with the application container, instantiating one-to-one communication within the services included in the mesh. Sidecars usually support only a restricted set of protocols, such as HTTP and gRPC. For example, in a Kubernetes cluster, a sidecar runs together every service and acts as a proxy for all traffic to and from the service. The Kubernetes framework itself manages the sidecar communication among other sidecars. The example architecture in Fig. 1 generalizes many existing solutions presented in Section 5.

So, developers can create microservices concentrating on implementing *business logic*, achieving bet-

ter transparent service-to-service communication between the applications and the end users. In addition, managing traffic, implementing observability, and ensuring security through control plane configuration and proxies enable service administrators to optimize performance and reduce costs while maintaining good separation of concerns. Moreover, a Service Mesh offers application developers greater flexibility and control in testing and releasing new functions or services in production, enhancing their CI/CD workflows [5]. For example, Service Mesh supports all the capabilities for managing and facilitating the correct transition among service versions.

Traffic Management

Service Meshes provide transparent communication in a microservice application through the traffic management capabilities of its control plane and proxies. Services do not need to know how to connect to another service because it is the Service Mesh that manages **service discovery, traffic routing, orchestration, and communication scaling** throughout the whole deployment.

In general, the specification of policies via APIs in the control plane allows administrators not only to route and load balance traffic but also to perform specific test mechanisms to prove the overall resiliency of applications.

The sidecar proxy works for the main application container, intercepting network requests and forwarding them to the appropriate backend services via the control plane definitions. Among the most relevant mechanisms employed to perform traffic management, we can name circuit breakers, timeouts, and retries. These features are required to implement the testing mechanisms for deploying different versions of the microservices, like A/B testing, Canary, and chaos engineering [2].

Observability

Service Meshes enable observability by offering **distributed logging, metrics gathering, and tracing** through sidecar proxies. The whole observability data is collected transparently from the sidecar network without requiring external services or traffic redirection to aggregate the data. The collected information provides insight into connected services and flowing traffic, immediately identifying possible bottlenecks or service downtimes. Observability helps identify issues, understand dependencies, and optimize application behavior, by enhancing reliability and user experience.

Proxies distributed next to microservices enable request tracking over multiple services, by allowing distributed tracing and logging abilities and providing insights into request latency and the topology of the communication between microservices. Consequently, collecting observability data within sidecars leads to data dissemination across the entire application. If the application is distributed across multiple sites, such as in a continuum of cloud resources, that produces two different outcomes: on the one hand, it enables immediate data processing; on the other hand, transferring and aggregating the data can introduce communication overhead.

Security

Service Meshes provide the mechanisms and practices for the entire security needs of microservices and their interactions. A Service Mesh can ensure secure communication between services by implementing **encryption, authorization, and authentication** for microservices.

Encryption is achieved through mutual Transport Layer Security (mTLS) between services by securing service-to-service communication. The Service Mesh can also manage microservice interactions with each other by enforcing authorization with access control. So, it can enable fine-grained policies that specify which services are allowed to communicate, with stricter control over the exchanged packets.

Authorization can extend to both ingress and egress at the network perimeter to provide a secure-by-default option without requiring any additional application code or infrastructure changes.

Moreover, the Service Mesh enforces authentication with certification authorities in combination with mTLS to ensure access to sensitive data only for verified users and services. By continuously monitoring traffic for anomalies, Service Mesh enables security-designed DevOps with rapid identification and response to potential security threats.

Service Meshes greatly alleviate the burden for developers and administrators of managing the whole infrastructure concerns of the application by introducing flexible management and control over a large number of services.

At the same time, Service Meshes architecture **tightly couples** microservices **in time and space** because of the use of proxies that facilitate communication by imposing only synchronous and blocking interactions. Due to the required synchronicity, this strict time coupling can hinder the scalability of microservices and their communication [2], [7].

Event Mesh

The distribution of devices, services, and resources across the cloud continuum requires increasing **asynchronicity in communication** without imposing strict response-time constraints. That enables scalability and compensates for lower reliability of devices and services executing in asynchronous environments. To address these challenges, transitioning from synchronous service-based architectures to robust asynchronous, event-driven models is essential to adapt the Service Meshes for large cloud continuum environments [8].

Event Mesh becomes an alternative solution to Service Mesh to overcome all these problems because of **asynchronous event-driven communication** among different services [9].

Event Mesh deploys both synchronous and asynchronous sidecar proxies, along with services and sensors, to enable applications to communicate by using different protocols and strategies. In the Event Mesh architecture of Fig. 2, services and heterogeneous data sources seamlessly interact among themselves through *event* exchange, where an *event* represents the interaction among microservices or either created or updated data. These events are published and consumed by microservices via *Brokers*, that act as Message Oriented Middlewares in the communication.

Event Mesh seamlessly connects brokers in the configuration and support of a network spanning various cloud platforms, edge Computing sites, and end devices. The event generated in the proxy is pushed to its designated broker, which acts as a MOM for the whole deployment in collaboration with other brokers. *Event Routers* route events to the proper destination. These routers consist of processes that share an updated knowledge of the topology, or a fraction of it and are capable of filtering and routing traffic. The *Subscription Manager* manages the propagation of subscriptions to topics and instruments brokers to accept incoming connections from sidecar proxies. Finally, the Event Mesh makes a common shared *Event Schema Registry* available across the whole infrastructure to enable event serialization and deserialization from different formats and programming languages.

Traffic Management

Event Meshes decouple service communication in space and time via **asynchronous message-oriented middleware** to obtain flexibility and scalability. MOM treats messages as self-contained units that can be stored, routed, and transformed while moving from sender to receiver. Message storage supports asynchronous communication to enable clients to perform other tasks while retaining messages until they have been retrieved and delivered. Message retention facilitates the implementation of any required delivery semantics, such as at least once, at most once, or exactly once [9].

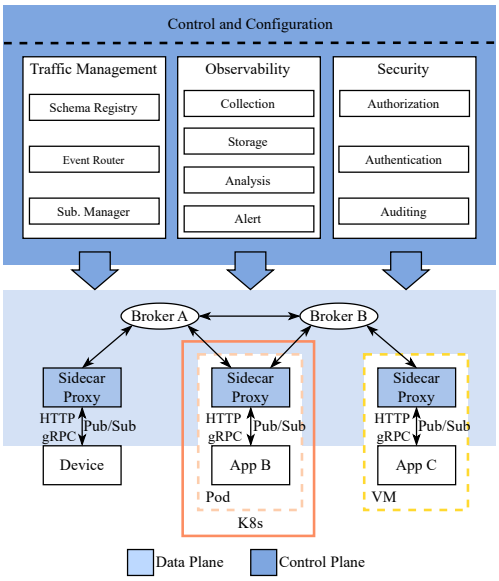

FIGURE 2. High-level abstraction of Event Mesh architecture, presenting the communication in the data plane with services, devices, and virtual machines, and of the pivotal functionalities in the control plane

Instead of routing messages based only on addresses, MOM brokers can rely on topic subscriptions, so creating a many-to-many communication pattern where publishers send messages to topics that multiple subscribers can access. MOM can also include additional routing logic of either broadcast or multicast for efficient distribution. Moreover, MOM can modify messages to meet the needs of either senders or recipients and provide tools for transformation through the *Event Schema Registry*.

In addition, MOMs easily scale to handle increasing data traffic and ensure reliable message delivery, even during network failures. For example, brokers can implement traffic load balancing for the mesh with queue-based messaging and can enable fault tolerance in case of service failures with asynchronous communication. Event brokers can also ease functions like canary deployments by duplicating the traffic directed to the new service version [10]. That approach does

not require distributed consensus among brokers as it relies on the local control policy.

All these benefits of asynchronous communication of Event Mesh are crucial in service chaining: in service compositions, the broker can facilitate the implementation of communication policies to preserve messages when recovering from failures. That capability also facilitates message retransmission at the application level when not provided by communication protocols.

Observability

Event Meshes enhance the collection of data for observability through **brokers and sidecars** that provide deeper insights into the communication state of the system.

The observability of Event Mesh depends on the ability of the sidecar proxy to intercept local signals from microservices. Due to the introduction of brokers, metrics regarding broker-to-broker communication are particularly relevant to assess the communication status. In particular, when the observability capabilities of microservices need to be extended to integrate typical metrics of data sources, also brokers most be modified to gather traces, then, the Event Mesh can easily transport observability information. For example, the **transparent exchange** of both communication data and event streaming from sensors, through the deployment of brokers, extends the capability of Event Mesh in gathering knowledge about the system.

Security

Event Meshes implement a zero-trust architecture with an asynchronous, non-blocking model where brokers play a central role in granting security features - such as authentication, authorization, and auditing. That allows to maintain security and trust in communication, to safeguard sensitive data from unauthorized access, and to ensure that messages remain confidential and intact throughout their journey.

Security shifts from an everyone-to-everyone trust model to a **one-to-one trust model**, where each service proxy authenticates with the broker. In Event Mesh, interactions are based on the submission of messages to various topics, so both exchanged data and services taking part in the mesh should be authenticated. As in Event Mesh, authorization is managed for event streams of data; the service must access to that stream and can read messages coming from multiple services published on that stream.

Event Mesh allows for a real asynchronous layer very suitable for cloud continuum applications by decoupling communication in both space and time, and deploying brokers communicating with sidecar proxies in each microservice.

MOM represent also a point of flexible control to decouple service-to-service communication and to shift from one-to-one to **many-to-many communication** within the mesh. Even if brokers enable more interoperability for devices, IoT and services, and integrate multiple communication protocols, we need to stress that can also cause **overhead in latency and resource consumption** due to both brokers and proxies.

Widespread Mesh Products

In recent years, the adoption of Service Mesh technologies has increased, leading to various open-source platforms with novel architectural approaches. We aim to present the most active and adopted open-source projects supported by the Cloud Native Computing Foundation (CNCF) [11], including **Istio**, **Linkerd**, **Consul**, and **Kuma** [12]. For Event Mesh, we analyze **Apache EventMesh** [10], **Knative Event Mesh** [13], and **Aeraki** [12], a hybrid solution combining both mesh principles.

We focus on key differences in the adopted proxy, data plane, control plane, and multicluster support across these platforms as they impact performances, compatibility, and deployment scenarios. The CNCF promotes Envoy Proxy as the standard sidecar proxy implementation in Service Mesh, adopted by *Istio*, *Consul*, and *Kuma* [12], to foster consistency across projects. In contrast, *Linkerd* uses its own proxy, optimized for resource efficiency and traffic management.

The data plane in these projects varies in supported protocols, significantly impacting the microservice application supported by a Service Mesh project. *Linkerd* supports TCP, HTTP, and gRPC, while Envoy, used by *Istio*, *Consul*, and *Kuma*, works with HTTP and gRPC and can interact with databases for traffic sniffing and filtering, including functioning as a Redis proxy for cluster availability.

Control planes differ in architecture and deployment models, by affecting the distribution of the application in edge-cloud sites. The *Istio* control plane is complex, supporting primary and remote clusters or a single external control plane. *Linkerd* has a simpler control plane with a controller for proxy configuration and high availability through component replication. *Consul* follows a client-server model, where *Consul* Servers manage cluster states and clients ask for service registration, health checks, and service discovery operations. *Kuma* offers Kubernetes-based and universal

control planes depending on the supported platform.

Istio supports multi-cluster setups with two models: Single Control Plane, propagating configurations across clusters, and Multi-Primary, where clusters share a federated control plane. *Linkerd* uses a centralized API server for multi-cluster communication. *Consul* excels in multi-datacenter and multi-region environments with WAN federation and distributed consensus achieved through the gossip protocol. *Kuma* provides multi-cluster support with a centralized control plane managing proxies in each cluster.

The application execution environment directly affects the choice of a Service Mesh project. *Istio* and *Linkerd* build on Kubernetes and containerized environments, with complex mesh configurations for VM or bare metal workloads. In contrast, *Consul* and *Kuma* offer easier management for multi-environment applications.

For Event Mesh, *Apache Event Mesh* uses Event Proxies for protocol compatibility (e.g., MQTT, Kafka), while *Knative* Event Mesh is proxyless, connecting event senders and brokers via event ingress and egress. *Aeraki* integrates the Envoy proxy, supporting advanced traffic management and event-based messaging. Event Mesh control planes also differ: Apache EventMesh manages MOMs, connectors, and schemas, *Knative* focuses solely on broker configuration, and *Aeraki* deploys the *Istio* control plane. *Apache Event Mesh* enables multi-cluster event bridging, supporting heterogeneous environments (cloud, edge, on-premise), while *Knative* and *Aeraki* are Kubernetes-centric, handling events from microservices.

Future Developments

Expanding Service and Event Mesh adoption to a broader range of applications requires performance improvements, compatibility with various execution environments, and standardization.

In Service and Event Mesh, sidecar proxies play a crucial role in traffic management, observability, and security, but they introduce overhead. A new approach emphasizes secure communication at the transport and network layers, using proxies only for enhanced features. The **Ambient Mesh** paradigm [14] employs **Zero Trust communication** at the transport layer using TCP and mTLS protocols and deploying a single proxy per node. **Cilium** adopts a **sidecarless model**, connecting services through TCP sockets with in-kernel code execution [15], [16]. **Network Service Mesh** [17] proposes an IP mesh that enables Zero Trust communication and deploys sidecar proxies only when combined with a Service Mesh implementation.

Mesh technologies must support diverse execution environments, as microservice applications run across various platforms. While most projects focus on multi-cluster environments, only *Consul* [12] and *Apache Event Mesh* manage an execution environment-agnostic control plane. The same also appears in the data plane, where services may use different isolation technologies, requiring proxies to accommodate them. *Istio*, for instance, integrates WebAssembly modules to support multiple isolation technologies beyond virtual machines and containers [12].

The differences in protocols adopted for proxy-to-proxy and broker-to-proxy communication can cause incompatibility within microservice applications. A key future direction involves standardizing proxy and broker interfaces using common formats like CloudEvents [10], [13]. That standardization will enhance interoperability and speed up the adoption of mesh technologies in microservices.

Conclusions

This paper examines advancements in communication management for cloud-based distributed applications, focusing on Service Mesh and Event Mesh approaches. It analyzes Service Mesh capabilities and introduces Event Mesh for asynchronous, event-driven communication among microservices in the cloud continuum.

We believe mesh approaches will impact microservice-based applications that span multiple scenarios as they supply functions for managing a wide range of infrastructure concerns. Industry 5.0 stands to gain significantly from zero trust security features that protect inter-service communication and recent advancements in low-latency traffic management at lower stack layers. Smart city applications, typically executed across distributed nodes, will benefit from holistic observability services and the multicluster capabilities of Event Meshes. Next-Generation Networks will leverage integrated observability to monitor network status, enhanced security for managing authentication and authorization in disaggregated environments, and advanced traffic management to optimize routing across various deployments.

Introducing a mesh in all these scenarios will enhance the role of infrastructure engineers, who will become responsible for managing infrastructure concerns of microservice-based applications.

ACKNOWLEDGMENTS

The European Union supported this work under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, “SEcurity and RIghts In the CyberSpace” (PE00000014 - program “SERICS”) CUP:J33C22002810001.

REFERENCES

1. Y. Wang, H. Kadiyala, and J. Rubin, “Promises and challenges of microservices: an exploratory study,” *Empirical Softw. Engg.*, vol. 26, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09910-y
2. W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, “Service Mesh: Challenges, State of the Art, and Future Research Opportunities,” in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 122–1225.
3. A. Koschel, M. Bertram, R. Bischof, K. Schulze, M. Schaaf, and I. Astrova, “A Look at Service Meshes,” in *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*, 2021, pp. 1–8.
4. Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, “Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review,” *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4213–4242, 2023.
5. M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan, and E. Verplanke, “Challenges and Opportunities in Performance Benchmarking of Service Mesh for the Edge,” in *2021 IEEE International Conference on Edge Computing (EDGE)*, 2021, pp. 78–85.
6. D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, “eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond),” *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023.
7. X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, “Dissecting Service Mesh Overheads,” 2022.
8. D. Gannon, R. Barga, and N. Sundaresan, “Cloud-Native Applications,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017.
9. H. Schaffner, “Comparing and contrasting services mesh and event mesh | solace resources,” https://solace.com/resources/white-papers/wp-download-comparing-and-contrasting-service-mesh-and-event-mesh, 2021.
10. A. S. F. Authors, “Event Mesh Documentation,” https://eventmesh.apache.org/, 2024.
11. L. Foundation, “Cloud native computing foundation, CNCF,” https://www.cncf.io/, 2015.
12. C. N. C. Foundation, “CNCF Documentation on Service Mesh,” https://landscape.cncf.io/guide#orchestration-management--service-mesh, 2024.
13. K. Authors, “Knative Code and Documentation,” https://knative.dev/docs/eventing/event-mesh/, 2024.
14. I. Authors, “Istio Ambient Mesh Documentation,” https://istio.io/latest/blog/2022/introducing-ambient-mesh/, 2024.
15. L. Foundation, “extended Berkeley Packet Filter, eBPF,” https://ebpf.io/what-is-ebpf/, 2021.
16. M. R. S. Sedghpour and P. Townend, “Service Mesh and eBPF-Powered Microservices: A Survey and Future Directions,” in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 176–184.
17. B. Dab, I. Fajjari, M. Rohon, C. Auboin, and A. Diquélou, “Cloud-native Service Function Chaining for 5G based on Network Service Mesh,” in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–7.

Andrea Sabbioni (andrea.sabbioni5@unibo.it) earned his M.Sc. in Computer Science Engineering cum laude from the University of Bologna, Italy, and is currently an Assistant Professor (RTD-A) there. His research interests include cloud, fog, and serverless computing, as well as middleware.

Sofia Montebugnoli (sofia.montebugnoli3@unibo.it) received her M.Sc. in Computer Science Engineering cum laude from the University of Bologna, Italy, and is currently pursuing a Ph.D. in Computer Science and Engineering at the same institution. Her research focuses on 5G architectures and cloud technologies.

Luca Foschini [SM] received a Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 2007, where he is now Full Professor. His interests span from integrated management of distributed systems and services to wireless pervasive computing and scalable context data distribution infrastructures and context-aware services.

Antonio Corradi [SM’19] graduated from the University of Bologna, Italy, and received an M.S. degree in electrical engineering from Cornell University, USA.

He is currently Full Professor of computer engineering at the University of Bologna. His research interests include all aspects of cloud and middleware and novel aspects of Cloud Continuum interactions.