CAMINO: Cloud-native Autonomous Management and Intent-based Orchestrator

Konstantinos Antonakoglou, Ioannis Mavromatis, Saptarshi Ghosh, Mark Rouse, Konstantinos Katsaros
Digital Catapult, London, UK
Emails: {konstantinos.antonakoglou, ioannis.mavromatis, saptarshi.ghosh, mark.rouse, kostas.katsaros}@digicatapult.org.uk

***Abstract*—This paper introduces CAMINO, a Cloud-native Autonomous Management and Intent-based Orchestrator designed to address the challenges of scalable, declarative, and cloud-native service management and orchestration. CAMINO leverages a modular architecture, the Configuration-as-Data (CaD) paradigm, and real-time resource monitoring to facilitate zero-touch provisioning across multi-edge infrastructure. By incorporating intent-driven orchestration and observability capabilities, CAMINO enables automated lifecycle management of network functions, ensuring optimized resource utilisation. The proposed solution abstracts complex configurations into high-level intents, offering a scalable approach to orchestrating services in distributed cloud-native infrastructures. This paper details CAMINO's system architecture, implementation, and key benefits, highlighting its effectiveness in cloud-native telecommunications environments.**

***Index Terms*—Intent-driven, Cloud-native, MANO, Observability, Configuration-as-data**

I. Introduction

The cloud computing and telecommunications industry stakeholders face increasingly complex and multifaceted challenges in the management and orchestration (MANO) of service and network intents, particularly as deployments become more heterogeneous in terms of underlying technologies and system scale [1]. The expectations for autonomous systems with automation capabilities are growing, necessitating closed-loop control at different levels and phases of MANO workflows.

To enable such capabilities, Network Function Virtualisation (NFV) MANO platforms allow services to consume computing and network resources (both physical and virtual) offered by infrastructure providers (InPs). These resources are distributed across distinct administrative domains (e.g., belonging to network operators) to support the deployment of interconnected meshes of Virtual Network Functions (VNFs) and Cloud-native Network Functions (CNFs).

According to ETSI Zero-touch Network and Service Management (ZSM) specifications [2], a management or administrative domain is an autonomous entity that groups computing and network resources under a single administrative authority. These domains are separated due to differences in infrastructure, data ownership, policies, and operational constraints, each responsible for resource management and security. They may face unique complexity and scalability challenges while operating with varying infrastructure capabilities, such as Radio Access Technologies or computing resources within the edge-cloud continuum.

Within a single administrative domain, a MANO platform orchestrates network functions and other services in a federated (vertical or horizontal) fashion, interacting with the domain components and external consumers of the northbound/southbound APIs. This becomes particularly complex when multiple administrative domains need to be orchestrated in parallel (e.g., as in [3]) or when multiple network segments or "edges" (each potentially with its own orchestrator) are part of the same administrative domain (e.g. as in [4]).

Based on the above, we present Cloud-native Autonomous Management and INtent-based Orchestrator (CAMINO), a scalable MANO architecture that enables the orchestration of services within multi-level administrative domains. We build upon traditional NFV MANO principles and extend them across a modernised architecture that can accommodate the requirements of a 6G and cloud-native system. Compared to other platforms, it facilitates a configuration-centric approach of zero-touch provisioning of deployment intents by enabling closed-loop multi-edge orchestration featuring service brokering, resource monitoring and admission control capabilities. In this paper, we focus on the architecture of CAMINO, the design decisions and the initial implementation of the idea, discussing the lessons learned, the challenges faced, and the enhancements provided compared to existing solutions.

The remainder of this paper is as follows. Sec. II presents related work on MANO platforms, highlighting existing approaches and limitations. Sec. III introduces the CAMINO architecture, detailing its functional components and their interactions. Sec. IV describes the implementation of CAMINO, including the tools and technologies used. Finally, Sec. V concludes the paper and outlines potential future directions.

II. Related Work

The increased complexity of cloud and edge deployments has led to the creation of MANO platforms that abstract and automate the management and configuration of such technologies. In [5], the survey underscores the value of a declarative approach to network slice provisioning with the translation of high-level service requirements into technical descriptions of network slices or subnets capable of fulfilling them. It is also noted that there is a lack of guidelines on how declarative provisioning APIs are designed. CAMINO builds upon this principle by decoupling the configuration values from the application code and the declarative deployment files, building upon the Configuration-as-Data (CaD) concept.

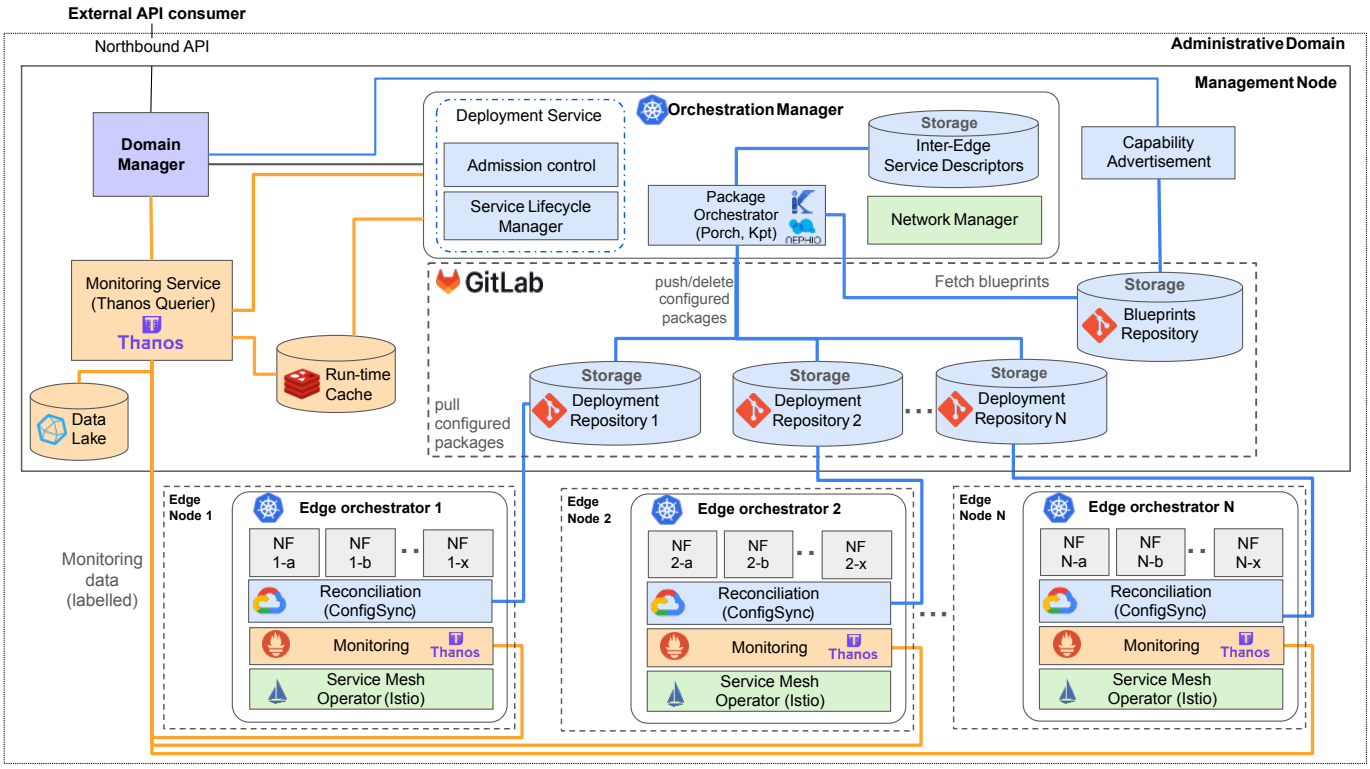

Fig. 1. Functional block diagram of CAMINO showing service orchestration (blue), monitoring (orange) and network management (green) elements.

Two established open-source end-to-end (E2E) MANO platforms are the Open Network Automation Platform (ONAP) [6] of Linux Foundation and the Open Source MANO (OSM) [7] from ETSI. They both support cloud infrastructure platforms such as OpenStack and Kubernetes to deploy VNFs and CNFs. However, both solutions have inherent complexity from legacy telecom use cases and ETSI standards, and neither ONAP nor OSM supports a declarative way of service provisioning [8]. CAMINO brings a modular architecture, building upon declarative and intent-driven models and provides a solution that moves beyond traditional NFV, aiming to manage modern multi-domain-multi-edge network resources and services.

Another Linux Foundation project that follows the above principles is Nephio [9], an intent-based cloud-native platform for the automated deployment and management of network functions based on CaD. However, Nephio is not an E2E MANO platform but acts as an abstraction layer between the E2E MANO platform layer and the layer of edge-level orchestrators/controllers. Furthermore, Nephio is a set of interchangeable open-source components per the CaD and intent-based orchestration principles. CAMINO utilises Nephio, leveraging how it handles configuration data and extends it with a more robust and scalable management and orchestration toolchain that allows for better lifecycle management (LCM) of service and network functions.

Even though the platforms above strive to provide holistic MANO solutions, they still fail to adequately address several critical administrative domain activities such as infrastructure management, SLA management and assurance, optimised placement and scaling, and policy enforcement, requiring additional components or extensions to bridge these gaps. The authors of [8] propose an architecture for zero-touch service MANO with run-time mechanisms that support it, but does not address scalability in VIM configuration management. They also focus on VM-based deployments such as OSM.

Authors in [10] highlight the importance of transitioning from VM-based to CNF-based workloads, introducing a Kubernetes Operator plane where multiple custom Kubernetes Operators declaratively implement MANO functionalities. However, this work does not focus on scalability and configuration management.

Finally, [11] focuses on managing cloud-native infrastructures using a Logical Kubernetes Cluster and scheduling cloud-native workloads but without addressing the required data structures required to orchestrate the deployments and the coordination with other similar platforms. CAMINO holistically addresses the above issues and provides a uniform solution envisioning to modernise MANO solutions in future networking systems.

III. System Architecture

Fig. 1 shows the functional blocks of the CAMINO architecture and their interconnections within the same administrative domain. The overall system enables the management and deployment of network function workloads in multiple *Edge Orchestrators*, monitoring resources for evaluating running deployments and admission control of deployment intents.

The edge orchestrators are controlled by the *Orchestration Manager*. We consider cloud-native Edge Orchestrators as well

as a cloud-native Management node. For service and network deployments across administrative domains, a cross-domain orchestrator (CDO) is required. The following sections describe CAMINO's function blocks, highlighting configuration details, design decisions, and relevant interconnections.

Listing 1. An example service deployment intent (JSON) on a single administrative domain (Domain-X) considering a network function chain with a CNF placed in a different administrative domain (Domain-Y).

```
{
  "domain_name": "Domain-X",
  "deployment_id": "338d10a2-2669-46e1",
  "timestamp": "2025-01-24T20:55:50.991211",
  "services": [
    {
      "package_name": "CNF-1",
      "version": "v1",
      "qos_level": "default"
    },{
      "package_name": "CNF-2",
      "version": "v3",
      "qos_level": "default",
      "dependencies": [
        {
          "after": "CNF-3",
          "domain": "Domain-Y",
          "fqdn": "yyy.yyy.yyy.yyy"
        },{
          "after": "CNF-1",
          "domain": "Domain-X",
          "fqdn": "xxx.xxx.xxx.xxx"
        }
      ]
    },{
      "package_name": "CNF-4",
      "version": "v2",
      "qos_level": "default",
      "dependencies": [
        {
          "after": "CNF-2",
          "domain": "Domain-X",
          "fqdn": "xxx.xxx.xxx.xxx"
        }
      ]
    }
  ]
}
```

A. Deployment Intents and Packages

A deployment intent represents an E2E service request deployed across multiple edge clusters (e.g., due to latency or privacy constraints of the E2E application). Listing 1 provides an example deployment intent. In our architecture, this E2E deployment intent comprises one or multiple service components, referred to as *packages*. These may include VNFs, CNFs, or other cloud-native components and configurations that collectively fulfil specific network and computing tasks. For example, a *package* may be an Nginx ingress controller, while another may define an Ingress rule (e.g., in a Kubernetes cluster). A deployment intent is a high-level description of all the services and network configurations, including their associated package names, relevant resource requirements and dependencies.

To manage deployment intents efficiently, we adopt the CaD principle, treating packages as *declarative configuration bundles* managed by version control systems [12]. This allows centralised and simplified tracking of changes, package versioning and collaborative authoring, with the benefits of a declarative approach that separates configurations from the code that operates them. The configurations are divided into blueprint ("dry") configurations, which contain the component assembly information, and deployment ("hydrated") configurations, that is, the declarative statement of the desired state. Consequently, we classify the repositories that host packages into *blueprint repositories* and *deployment repositories*, respectively.

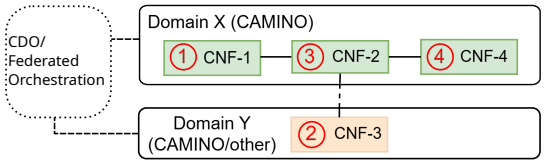

Fig. 2. Example deployment of a network function chain based on Listing 1, and the order of deployment (in red), between two administrative domains.

B. Order of Deployment and Service Brokering

Listing 1 shows the deployment order and interdependencies across different network and service functions. The order and interdependencies can either be chosen by an end-user or automatically generated as part of a service brokering. For example, such a deployment intent can originate from a user portal or a cross-domain orchestrator (as in [13]).

In Fig. 2, and according to Listing 1, we see the execution order for four CNFs, three deployed by CAMINO in the administrative Domain X and one handled by a different administrative authority in Domain Y. The order dictates the generation of the required deployment configuration files for the entire chain. This is crucial because specific configurations (e.g., a Kubernetes "service", or the FQDN of a different domain) must be used as inputs to the following packages. Since JSON captures associations without order, we adopt a "labelling" approach, using the "after" label to control the deployment sequence. Based on the given labels, the Domain Manager performs a basic topological ordering and constructs a directed graph of the services with linear ordering. This graph represents the order of execution.

C. Reconciliation and Deployment Configuration

Based on the above order, the blueprint configurations describing a package are modified to compile a *deployment configuration*. These configurations are manifests stored in the corresponding deployment repositories. Each repository is linked to a *reconciliation operator* installed in each edge node (e.g., Kubernetes cluster). This operator synchronises the state of an edge deployment with the contents of its assigned deployment repository, which serves as the source of truth. In principle, when a configuration is added to or updated in a repository, the edge orchestrator automatically deploys it. Similarly, when a configuration is removed, the service is terminated.

To automate the declarative distribution and customisation of one or more blueprint packages, an Inter-Edge Service Descriptor (IESD) is required, stored in a separate storage entity. The IESD bundles the minimum deployment requirements and additional configurations to compile new manifests. All blueprint configurations include various parameterised field values and act as templates, enabling CRUD operations during deployment without violating the declarative manifest's schema.

For example, if an intent requires a package to be deployed, a JSON document as in Listing 2 is created and used to generate the deployment configuration. If the

deployment corresponds to a Kubernetes Deployment resource, the "version" and "package_name" labels can generate the container image tag, the resources can be used as part of the Kubernetes resource request or limits (i.e., in *spec.template.spec.containers[].resources*), while the "qos" label can control an environmental variable that modifies a set of exposed QoS configurations from the application. Additionally, different labels can be included (e.g., network resources, network slices, specific naming conventions, etc.) depending on application requirements.

Listing 2. Description of a package instance in JSON format

```
{
  "name": "example_package",
  "package_requirements": [{
    "qos": "default",
    "revision": "v5",
    "package_resources": {
      "container": "example_container",
      "cpu": 8,
      "memory": "1000000Ki"
    }
  }]
}
```

D. Orchestration Manager

Each of the packages stored in the blueprints repository is a potential service or configuration that can be deployed by the Orchestration Manager once a deployment intent is forwarded by the Domain Manager. The Orchestration Manager is a logical entity composed of subcomponents that provide the following capabilities:

- Registration/deletion of blueprint and deployment repositories.
- Service LCM and distribution of configuration bundles from blueprint repositories to deployment repositories.
- Manipulation of blueprint packages, including composition or selection of the appropriate IESDs.
- Configuration of network resources consumed by services of each deployment.
- Discovery of minimum requirements for selected intents to enable admission control using the monitoring data.

E. Admission Control and Resource monitoring

The role of the Admission Controller is twofold. Initially, it checks the configurations and IESDs (validates the schema or the content of the files) to prevent misconfiguration. Moreover, it verifies whether the required resources are available for a service (e.g., enough CPU cores and RAM) or a configuration (e.g., an Ingress Controller is available).

A monitoring plane is required to track resource utilisation. The monitoring tools at the edge clusters collect the available resources and monitor the existing deployments. The monitoring service at the management node ensures enough resources are available at each cluster, raises alerts for misbehaving applications and aggregates the monitoring data for visualisation purposes.

There are two kinds of resources being monitored: 1) **Infrastructure resources**: overall metrics of each cluster within the administrative domain. 2) **Workload resources**: metrics specific to each deployed network function.

External consumers (e.g., a cross-domain orchestrator [13]) can also request access to monitoring data through the Domain Manager APIs. These requests are subject to administrative policies (e.g., providing only aggregated infrastructure utilisation instead of edge-level information). The Admission Control and Service Lifecycle Manager are responsible for deploying the configurations across the various edge nodes.

Listing 3. Description of a network deployment intent extracted from Listing 1

```
{
  "deployment_id": "338d10a2-2669-46e1",
  "services": [
  {
    "name": "CNF-1",
    "endpoints": [{
      "host": "svc1", "port": 80, "protocol": "HTTP"
  }],
    "links_to": [{
      "name": "CNF-2", "type": "intra-edge"
    }]
  },
  {
    "name": "CNF-2",
    "endpoints": [{
      "host": "svc2", "port": 80, "protocol": "HTTP"
    }],
    "links_to": [{
      "name": "CNF-1","type": "intra-edge"
    },{
      "name": "CNF-3",
      "type": "cross-domain",
      "resolution": {
        "domain": "Domain-Y", "fqdn": "yyy.yyy.yyy.yyy"
      }
    },{
      "name": "CNF-4", "type": "inter-edge"
    }]
  },
  {
    "name": "CNF-4",
    "endpoints": [{
      "host": "svc4", "port": 80, "protocol":"HTTP"
    }],
    "links_to": [{
      "name": "CNF-2", "type": "inter-edge"
    }]
  }]
}
```

F. Domain Manager

The Domain Manager serves as the only point of contact for external entities. It provides the following capabilities using its northbound API endpoints:

- Advertises the available services and configurations of the blueprint repository as well as the total reserved resources to trusted external users or higher-level orchestrators.
- Receives deployment and termination intents, translating the relevant segments of these requests to the APIs of the Orchestration Manager's service LCM and network management components.
- Creates the topological linear ordering for the deployment.
- Manages authorisation and authentication for northbound endpoint calls.
- Discovers external domain network information (e.g., FQDN) in cooperation with other Domain Managers.
- Queries monitoring data from the Monitoring service.
- Responds to external health-check requests.

G. Network Manager

The Network Manager entity enables a programmable communication fabric across services. Establishing the fabric begins after the Network Manager receives the connectivity configuration details extracted from the deployment intent. Such configuration may include topology directives, policy enforcement rules, failover mechanisms, and security constraints.

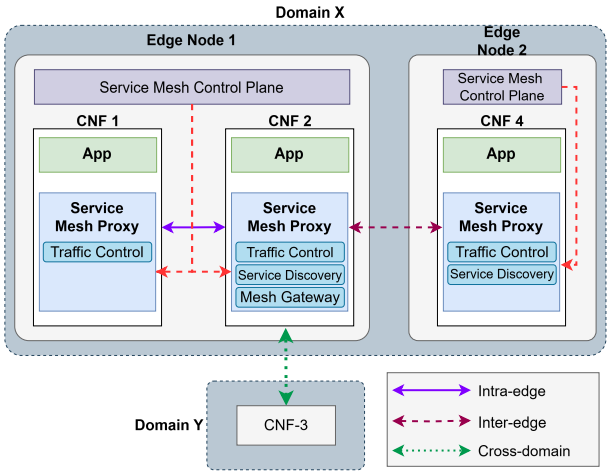

Fig. 3. Service Mesh deployment (as in Listing 1) of a network function chain placed between two different administrative domains.

CAMINO leverages the Service Mesh paradigm, which abstracts inter-service communication by providing a control plane that accepts high-level declarative definitions, translating them into low-level network configuration, and injecting them into a data plane that spans the target services. Such abstraction simplifies various connectivity configurations such as: 1) **Intra-edge**: The minimum required connectivity configuration within a single cluster. 2) **Inter-edge**: Connectivity across multiple clusters within the same network domain. 3) **Cross–domain**: Connectivity across different administrative domains using CAMINO or a compatible solution.

Upon receiving the deployment intent, the Domain Manager extracts the deployment information (i.e., the associated packages and topology) and forwards it to the Orchestration Manager. Then, the Network Manager generates and implements a service mesh configuration based on the brokering plan that distributes the services to Edge nodes. An example is found in Listing 3. This information is used to hydrate blueprint network configuration files stored in the edge deployment repositories and deployed in the edge clusters.

Fig. 3 illustrates an example service mesh deployment based on the deployment intent of Listing 1. Each Edge Node has its own Service Mesh Control Plane that injects and configures a Service Mesh Proxy in each deployed CNF. For this example, we assume that in Domain X, the Orchestration Manager assigns CNFs 1 and 2 to Edge Node 1, so the Network Manager needs to establish intra-edge communication between the two CNFs. CNF 4 is assigned to Edge Node 2; thus, inter-edge communication between CNF2 and CNF4 is required. Domain Y hosts CNF3, so cross-domain communication between CNF2 and CNF3 is necessary. For cross-domain discovery, the Network Manager can either receive the remote domain FQDN from an end-user, request this information from a trusted cross-domain orchestrator, or maintain a list of trusted domains within the Domain Manager.

IV. Implementation

This section describes the tools and solutions used or developed that enable E2E service deployments. In our setup, all nodes -both management and edge- are considered to be Kubernetes clusters with a MetalLB load-balancer, Calico as a Container Network Interface, and Nginx as the Ingress Controller. It is important to note that other solutions are acceptable, provided they comply with CAMINO's architecture.

A. Orchestration and Domain Management

The package repositories are managed by GitLab, chosen due to its open-source nature and privacy preservation (i.e., local deployment). Our package manager of choice is *kpt* [14], used for automated authoring and manipulation of Kubernetes Resource Model (KRM) packages, e.g. [15]. Kpt was selected over Helm as it natively integrates with CaD and GitOps workflows. Additionally, it treats upstream packages as immutable artefacts with "pull" and "customisation" operations being separate steps. This prevents local changes from being overwritten, making updates easier to manage.

Kpt introduces functions (e.g., setters) that enable CRUD operations on blueprint packages using IESD descriptors. All blueprint packages are described with parameterised, customisable fields in the form of comments within the defined YAML manifest. These comments dictate the parameters that can be manipulated such as metadata (e.g., namespaces, names, labels), spec (e.g., images, resources, selectors) and others. All parameters are managed with IESDs in the form of PackageVariantSets (PVSs), a custom resource (CR) managed by *porch* [9]. The blueprint packages are reusable and can become deployment packages through "hydration". In general, *porch* handles creating, deleting, and maintaining the blueprint and deployment package repositories within GitLab, registering all repositories across management and edge nodes, and generating and deploying porch-compliant CRDs.

The *Package Orchestrator* (developed in Python 3.11) wraps around kpt and *porch*, automating the creation and management of PVSs, kpt function manipulation, and the storage or retrieval of all packages. The *Domain Manager* (developed in Python 3.11) handles creating, deleting, and maintaining the blueprint and deployment package repositories within GitLab and triggers the registration of new repositories across all management and edge clusters. Upon receiving a deployment intent and generating a topological order, the intent JSON is sent to the *Deployment Service* (developed in Python 3.11). This service combines the *Service Lifecycle Manager* and the *Admission Controller* functionality. It handles package deployment proposals and approves them under certain conditions (e.g., if no misconfiguration exists, or if enough resources are available for a deployment). Misconfigurations are checked by "dry-running" the updated/hydrated packages and evaluating the response of the edge nodes. Resources are queried from the *Monitoring Service* (both workload and infrastructure resources) or directly from the Kubernetes API if it concerns cluster or network configuration resources.

A package approval triggers *porch* to push the package to a deployment repository. Later, the *Reconciliation Manager* of each edge node handles the package deployment using the local orchestrator. Similarly, when a termination intent is

received, the *Deployment Service* checks whether it is possible or if a conflict may arise and accordingly handles the termination, instructing *porch* to delete the deployment package, and the reconciliator terminates the Kubernetes deployment. As a reconciliation tool, ConfigSync [16] was chosen due to its edge-based architecture which supports horizontal scaling in the case of a large number of edge nodes, compared to other solutions, such as ArgosCD, that follow a server-client model.

B. Network Management

The *Network Manager* (developed in Python 3.11) implements the Service Mesh paradigm leveraging Istio Service Mesh [17]. Once a deployment is triggered by the *Deployment Service*, and if communication with another deployment is required, an Istio Envoy Proxy Sidecar is injected in each Pod. This enables traffic management by intercepting traffic and applying policy-based routing.

The *Network Manager* is responsible for generating all Istio CRDs, adhering to CaD principles and utilising pre-existing network blueprint packages. The CRDs created are similarly deployed alongside the Pods across all edge nodes and are translated by the Istio control plane into low-level Proxy configurations. To provide isolation across deployments, we group CNFs of a specific chain into Kubernetes Namespaces and create and apply network policies using and adjusting pre-existing blueprint packages. Finally, we consider three different connectivity configurations: 1) **Intra-edge**: Leveraging the Virtual Service (VS) and Destination Rule (DR) CRDs to implement network policies, such as load balancing. 2) **Inter-edge**: Istio's model for individual multi-cluster control planes simplifies inter-edge service discovery and enables high availability using the VS, DR and Service Entry CRDs for remote service reachability. Additionally, we follow the *Namespace Sameness* principle to identify remote services. 3) **Cross-Domain (External)**: Cross-network Control Planes enable remote service discovery via the East-West Gateway CRD, allowing clusters to expose local services on specific ports through the domain's External IP.

C. Monitoring

Finally, the monitoring capabilities are enabled by Thanos and Prometheus [18] instances in each Edge Node to capture performance data for the Edge Nodes and the services. Each Prometheus instance in each cluster hosts a Thanos agent as a sidecar. All agents are queried by a Thanos Querier instance controlled by the management cluster's *Monitoring Service* (developed in Python 3.11). The *Monitoring Service* requests monitoring data for each metric of interest from all Thanos agents with a single PromQL query. This data can then be passed to relevant data-consuming services such as the Orchestration Manager or the Domain Manager.

Monitoring data are separated with custom labels, which are assigned during the service deployment or cluster instantiation. For example, the Deployment service assigns pod names a unique identifier, and edge nodes have unique contexts and names. The labels are exchanged between the Orchestration Manager and the Monitoring Service using a run-time cache (Redis was chosen for its lightweight nature and stability). Finally, all monitoring data is stored in a data lake (using InfluxDB) for long-term analysis and visualisation.

V. Conclusion

This paper introduces CAMINO, a cloud-native, autonomous management and intent-based orchestration framework designed to address the complexity of service deployment across multiple edge nodes leveraging the CaD principle. Its service orchestration, network management and monitoring components enable scalable, zero-touch provisioning of heterogeneous services to meet the demands of 6G systems. The CAMINO framework enhances the efficiency and observability of complex edge-cloud deployments. Future work will extend its intent-based capabilities with AI-driven service management and cross-administrative domain orchestration. This research contributes to the advancement of autonomous network orchestration, paving the way for scalable, resilient, and self-managing telecommunications infrastructures.

Acknowledgements

This work is a contribution by Project REASON, a UK Government funded project under the Future Open Networks Research Challenge (FONRC) sponsored by the Department of Science Innovation and Technology (DSIT).

References

[1] T. Zhang *et al.*, "NFV platforms: Taxonomy, design choices and future challenges," *IEEE TNSM*, vol. 18, no. 1, pp. 30–48, 2021.
[2] ETSI Industry Specification Group (ISG), "Zero-touch network and Service Management (ZSM); Intent-driven autonomous networks," European Telecommunications Standards Institute (ETSI), Tech. Rep. GR ZSM 011 V2.1.1, Sep. 2024.
[3] S. Moazzeni *et al.*, "5G-VIOS: Towards Next Generation Intelligent Inter-domain Network Service Orchestration and Resource Optimisation," *Computer Networks*, vol. 241, p. 110202, 2024.
[4] G. Scivoletto *et al.*, "Development & Reliable Orchestration of Network Applications for the Automotive Domain Across the Edge-to-Cloud Continuum," in *Proc. of EuCNC/6G Summit*, 2024, pp. 1055–1060.
[5] P. Wyszkowski *et al.*, "Comprehensive Tutorial on the Organization of a Standards-Aligned Network Slice/Subnet Design Process and Opportunities for Its Automation," *IEEE Commun. Surv. Tutor*, vol. 26, no. 2, pp. 1386–1445, 2024.
[6] "ONAP," Linux Foundation. [Online]. Available: https://www.onap.org/
[7] "ETSI OSM," ETSI. [Online]. Available: https://osm.etsi.org/
[8] S. L. Correa *et al.*, "Supporting MANOaaS and heterogenous MANOaaS deployment within the zero-touch network and service management framework," *IEEE Commun. Mag.*, vol. 8, no. 2, pp. 4–11, 2024.
[9] "Nephio." [Online]. Available: https://nephio.org/
[10] A. Mohammadi and N. Nikaein, "Athena: An Intelligent Multi-x Cloud Native Network Operator," *EEE JSAC*, vol. 42, no. 2, pp. 460–472, 2024.
[11] T.-N. Nguyen *et al.*, "A Design and Development of Operator for Logical Kubernetes Cluster over Distributed Clouds," in *Proc. of IEEE/IFIP NOMS*, 2024, pp. 1–6.
[12] R. Bhagwan *et al.*, "Learning Patterns in Configuration," in *Proc. of IEEE/ACM ASE*, 2021, pp. 817–828.
[13] K. Katsaros *et al.*, "AI-Native Multi-Access Future Networks—The REASON Architecture," *IEEE Access*, vol. 12, pp. 178 586–178 622, 2024.
[14] "kpt: Automate kubernetes configuration editing." [Online]. Available: https://kpt.dev/
[15] "Nephio Example Packages," ETSI. [Online]. Available: https://github.com/nephio-project/nephio-example-packages
[16] "Configsync." [Online]. Available: https://github.com/GoogleContainerTools/kpt-config-sync
[17] "Istio," Linux Foundation. [Online]. Available: https://istio.io
[18] "Thanos monitoring." [Online]. Available: https://thanos.io/