# iDynamics: A Configurable Emulation Framework for Evaluating Microservice Scheduling Policies under Controllable Cloud–Edge Dynamics

Ming Chen*, Muhammed Tawfiqul Islam, Maria Rodriguez Read, Rajkumar Buyya, *Fellow, IEEE*

***Abstract*—Designing and evaluating microservice scheduling policies in cloud–edge environments is challenging because application behavior is driven by multiple interacting dynamics: call-graph topology changes with the mix of request types, traffic along service chains becomes highly imbalanced, and cross-node network conditions such as latency and bandwidth fluctuate over time. Deploying and instrumenting large geo-distributed testbeds to study these effects is costly, disruptive, and often not repeatable. Existing simulators and emulators typically focus on either infrastructure dynamics or application workloads, and rarely capture their joint impact on end-to-end performance while running real services on a production-grade orchestration stack. This paper proposes `iDynamics`, a configurable emulation framework that exposes these dynamics as controllable experimental factors while running real microservice benchmarks on a Kubernetes cluster with 46 virtual machine instance nodes. `iDynamics` mainly comprises three modular components, including Graph Dynamics Analyzer, Networking Dynamics Manager, and Scheduling Policy Extender. Experiments on the testbed show that `iDynamics` can accurately emulate targeted network conditions, track diverse call graph and traffic patterns, and help quantify how different scheduling policies mitigate performance bottlenecks under controllable and repeatable dynamics.**

***Index Terms*—Microservices, cloud–edge continuum, call graph dynamics, networking emulation.**

## I. Introduction

Microservice architectures have become the de facto style for building cloud and edge applications. By decomposing functionality into independently deployable services, they provide elasticity and rapid evolution of individual components. However, when tens or hundreds of microservices are deployed across a cloud–edge continuum, maintaining Service Level Agreements (SLAs) becomes non-trivial: user requests traverse long service chains, services are shared across applications, and traffic patterns vary significantly over time [1]–[3].

In such settings, scheduling policies—that is, the placement and migration of microservice instances onto available nodes—react to dynamics at three layers: (i) **workload dynamics**, such as changing mixes and proportions of request types; (ii) **application-level dynamics**, such as evolving call-graph structures and imbalanced traffic between pairs of services; (iii)**networking dynamics**, such as fluctuating cross-node delays and available bandwidth in the cloud–edge continuum. These intertwined effects directly impact the end-to-end performance of microservice applications deployed on Kubernetes [4] clusters.

Designing scheduling policies that deal with these dynamics already requires substantial effort; rigorously *evaluating* them is even harder. Deploying alternative schedulers in production-scale, geographically distributed clusters is costly, disruptive, and rarely repeatable. Purely simulation-based evaluations, in contrast, often abstract away essential aspects of the Kubernetes stack, the service mesh, and real microservice implementations, which limits fidelity. Therefore, there is a need for an evaluation environment that runs real services while exposing key sources of non-determinism as experimental knobs rather than as uncontrolled environmental factors.

A large body of work has addressed microservice management and scheduling, for example through service-level-objective (SLO)-oriented resource management [2], [5]–[7], trace-based call graph analysis [3], [8], [9], network dynamics emulation [10]–[12], and network-aware pod placement in Kubernetes clusters [13]–[16]. Complementary work has proposed microservice simulators and performance-testing tools that couple real orchestrators with synthetic workloads. These categories provide complementary capabilities but target different purposes: management systems and Kubernetes extensions implement particular decision logic, network emulators reproduce infrastructure behavior, and simulators trade live-system fidelity for controlled exploration. Kubernetes-in-the-Loop, for example, executes Kubernetes scheduling and autoscaling components within a microservice simulation [17]. Among the representative systems compared in Table I, however, no single framework combines live microservice execution, runtime call-graph and traffic observation, configurable node-pair network conditions, and a reusable interface for comparing scheduling policies under matched scenarios.

To fill this gap, we propose `iDynamics`, a configurable emulation framework for evaluating microservice scheduling policies on Kubernetes-based cloud–edge clusters under controllable dynamics. In this work, *controllable dynamics* denotes an experimental contract for configuring, observing, recording, and replaying the time-varying factors that influence microservice placement. At the workload and application layer, users specify request types, queries per second (QPS), mix proportions, and evolution schedule; the Graph Dynamics Analyzer observes the active runtime dependencies and edge traffic in call graphs triggered by those configured requests.

*Corresponding author.

Ming Chen, Muhammed Tawfiqul Islam, Maria Rodriguez Read, and Rajkumar Buyya are with the Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia (e-mail: mingc4@student.unimelb.edu.au; tawfiqul.islam, maria.read, rbuyya@unimelb.edu.au).

At the infrastructure layer, users generate or replay directed latency and bandwidth matrices and apply them through the Networking Dynamics Manager. Each experiment run records the traces, monitoring interval, initial service placement, constraints, and policy-specified SLA targets so that compared policies consume equivalent time-indexed inputs. However, key metrics, such as end-to-end response time, throughput, resource usage, and SLA violations, are still measured experimental results rather than directly controlled variables.

Instead of proposing yet another scheduling algorithm, `iDynamics` focuses on the *evaluation problem*: it instruments the service mesh, injects network dynamics, and provides an extensible policy interface so that both existing and new schedulers can be studied under repeatable conditions. The framework is implemented as a set of modular components, including the Graph Dynamics Analyzer, Networking Dynamics Manager, and Scheduling Policy Extender.

Our main contributions are as follows:

- We propose the notion of *controllable dynamics* for microservice scheduling evaluation and implement it across the call graph, traffic, and network layers in a cloud–edge cluster.
- We develop `iDynamics`, a modular emulation framework that combines (i) service-mesh-based reconstruction and analysis of dynamic call graphs, (ii) fine-grained cross-node delay and bandwidth emulation with distributed measurement, and (iii) a pluggable scheduling-policy interface with reusable utility functions.
- We conduct extensive evaluations, including service mesh overhead, call graph construction overhead, validation of dynamic networking conditions, continuous call graph dynamics, and different policy studies under mixed workload modes for representative microservice benchmarks.

The rest of the paper is structured as follows. Section II reviews related work. Section III discusses background and challenges. Section IV introduces the `iDynamics` framework and its main components. Sections V, VI, and VII detail the designs of the Graph Dynamics Analyzer, Networking Dynamics Manager, and Scheduling Policy Extender, respectively. Section VIII presents performance evaluation and case studies, and Section IX concludes the paper and outlines future work.

## II. Related Work

The dynamics of microservice-based systems in cluster and cloud–edge environments have been studied from multiple angles. We briefly review these directions and position `iDynamics` relative to them. Table I compares `iDynamics` with representative work using a capability-oriented coding rule: full support denotes a capability implemented or exposed by the proposed system itself, whereas partial support includes narrower, offline, manual, simulation-only, or evaluation-harness support. For clarity, the table groups systems by their primary role: microservice management and scheduling, network emulation, Kubernetes scheduling extension, or microservice simulation and in-the-loop evaluation. As these roles may overlap, the individual capability cells provide the cross-category comparison.

### A. Microservice Management and Resource Scheduling

FIRM [5] leverages online telemetry and machine learning models to localize SLO violations and resource contentions, then mitigates them through fine-grained reprovisioning actions. Erms [2] formulates scalable resource management models for shared microservices with large call graphs, assigning latency objectives to individual services. GrandSLAm [6] predicts completion times for stages of microservice execution and dynamically batches and reorders requests based on these predictions. Sage [7] uses probabilistic models to diagnose cascading QoS violations in interactive microservices at scale. TraDE [18] introduces a network- and traffic-aware adaptive scheduling framework to maintain QoS targets under changing workloads and network conditions. However, these systems demonstrate sophisticated management logic but treat their execution environment largely as given: they do not expose a generic evaluation framework that allows different scheduling policies to be plugged in and compared under configurable call graph, traffic, and networking dynamics.

### B. Network Dynamics Emulation

Network dynamics strongly influence microservice performance in cloud–edge systems. THUNDERSTORM [10] evaluates distributed systems under dynamic topologies using micro- and macro-benchmarks. Kollaps [11] provides a decentralized topology emulator, while SplayNet [12] builds on Splay [21] to emulate arbitrary overlay topologies. These tools are effective at reproducing network-level behavior but are primarily agnostic to microservice call graphs, SLAs, and scheduling policies. At the level of individual nodes, several works inject delays or shape bandwidth using Linux traffic control primitives [5], [22]–[25]. Most, however, either employ static or uniform delay matrices or apply configurations at the granularity of a whole node, which makes it difficult to study heterogeneous pairwise conditions in a cloud–edge cluster.

### C. Network-Aware Microservice Scheduling

Network-aware schedulers exploit infrastructure-level information when placing microservices. NetMARKS [13] uses service-mesh metrics from Istio [26] to place Kubernetes pods for 5G edge applications. Marchese et al. extend the default Kubernetes scheduler with network-aware placement policies [14], [15]. OptTraffic [16] optimizes cross-machine traffic for multi-replica microservices by migrating containers with strong dependencies. However, these works confirm the value of incorporating network metrics into scheduling decisions but typically introduce a specific policy and evaluate it against the default Kubernetes scheduler, often in static or lightly controlled environments.

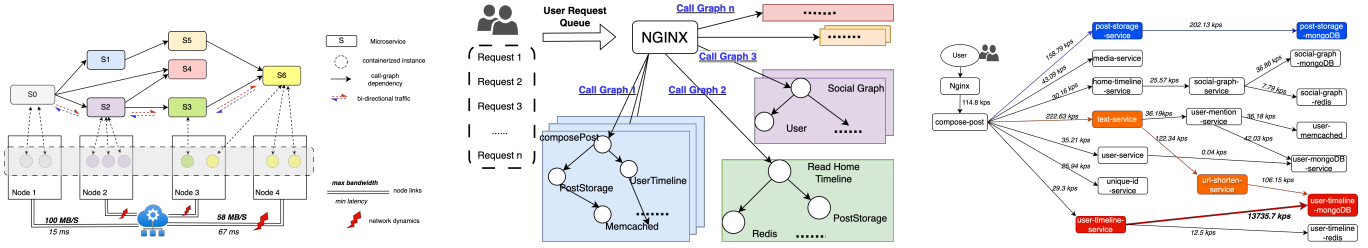
### D. Microservice Simulators and Evaluation Tools

Beyond schedulers and network emulators, several tools support simulation or emulation of microservice systems. Benchmark suites such as DeathStarBench [27] provide realistic microservice applications and workloads. $\mu$Bench [28] benchmarks microservice applications with dummy services

TABLE I
CAPABILITY COMPARISON OF IDYNAMICS WITH REPRESENTATIVE WORK.

| Work | Microservice Management | | | Workload–Dynamics Control | | | Call–Graph Dynamics Analysis | | | Network–Dynamics Control | | | | Scheduling–Policy Extensibility | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SLA/SLO Support | Placement/ Scheduling | E2E Metrics | Multiple Request Types | Mixed Proportions | Temporal Mix Modes | Dependency Discovery | Traffic Analysis | Graph Evolution | Emulation | Latency | Bandwidth | Measurement | Pluggable Interface | Utility Modules | Cloud–Edge Support |
| ***Microservice management and resource-scheduling systems*** | | | | | | | | | | | | | | | | |
| FIRM [5] | ✓ | ✓ | ✓ | ✓ | △ | △ | ✓ | △ | △ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Erms [2] | ✓ | ✓ | ✓ | △ | ✗ | ✗ | ✓ | ✗ | △ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| GrandSLAm [6] | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | △ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Sage [7] | ✓ | ✓ | ✓ | △ | ✗ | ✗ | ✓ | △ | △ | ✗ | ✗ | △ | ✓ | ✗ | ✗ | ✗ |
| ***Network emulation*** | | | | | | | | | | | | | | | | |
| Kollaps [11] | ✗ | ✗ | △ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | △ |
| THUNDERSTORM [10] | ✗ | ✗ | △ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | △ | ✗ | ✗ | △ |
| ***Kubernetes and network-aware scheduling extensions*** | | | | | | | | | | | | | | | | |
| NetMARKS [13] | △ | ✓ | △ | ✗ | ✗ | ✗ | ✓ | ✓ | △ | ✗ | ✗ | ✗ | ✗ | △ | ✗ | ✓ |
| Marchese et al. [14], [15] | △ | ✓ | ✓ | ✗ | ✗ | ✗ | △ | ✓ | △ | △ | △ | ✗ | ✓ | △ | ✗ | ✓ |
| OptTraffic [16] | ✗ | ✓ | △ | ✗ | ✗ | ✗ | △ | ✓ | △ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| TraDE [18] | ✓ | ✓ | ✓ | ✓ | △ | ✗ | ✓ | ✓ | △ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| ***Microservice simulators and in-the-loop evaluation tools*** | | | | | | | | | | | | | | | | |
| MiSim [19] | △ | △ | ✓ | ✓ | △ | ✓ | ✗ | ✗ | ✗ | △ | ✓ | ✗ | ✗ | △ | △ | ✗ |
| K8s-in-the-Loop [17] | △ | ✓ | ✓ | △ | △ | △ | ✗ | ✗ | ✗ | △ | ✓ | ✗ | ✗ | ✓ | △ | ✗ |
| CloudNativeSim [20] | ✓ | ✓ | ✓ | ✓ | ✓ | △ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| **iDynamics (This work)** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Legend:* ✓ = full support; △ = partial or implicit support; ✗ = not reported. Category bands identify each work's primary role.




(a) A workflow illustrating containerized microservice execution and communication under networking dynamics in a cloud–edge continuum.

(b) Different request types trigger distinct call graph topologies. (application source [27])

(c) A scenario of imbalanced traffic loads among UM–DM service pairs under a high workload, 5k QPS. (application source [27])

Fig. 1. Motivating examples for microservice scheduling in cloud–edge environments: (a) network dynamics in the cloud–edge continuum, (b) request-dependent call graph variability, and (c) imbalanced traffic distribution across UM–DM pairs.

on Kubernetes-based platforms. MiSim [19] models multiple service operations, time-varying workload generators, latency fault injection, resilience mechanisms, and extensible strategy implementations. More recently, CloudNativeSim [20] builds on CloudSim [29] to provide a discrete-event simulator for cloud-native, microservice-based applications, modeling service graphs, pods, and virtual machines and exposing policy interfaces for service management. Kubernetes-in-the-Loop [17] connects MiSim [19] to actual `kube-scheduler` and cluster-autoscaler implementations, thereby improving the fidelity of orchestration-policy experiments. Although this design supports alternative Kubernetes configurations and policies, the application execution and network effects remain within the simulation. These works provide essential building blocks and taxonomies but do not provide a framework that simultaneously (i) observes dynamic microservice call graphs, (ii) emulates heterogeneous cross-node network conditions, and (iii) exposes a pluggable interface for arbitrary scheduling policies while running real microservice containers on a Kubernetes-based cloud–edge cluster. `iDynamics` is designed to fill these gaps.

## III. BACKGROUND AND CHALLENGES

### A. Background

Microservice architecture decomposes an application into small, independently deployable services that communicate over the network [3], [27]. In cloud–edge computing, these services can be deployed in central cloud data centers, on edge nodes near end users and devices, or across both. In practice, an application may be partitioned so that some microservices run in the cloud (for global data, analytics, or coordination) while latency-sensitive components execute at the edge, communicating via RPCs or REST APIs.

This hybrid deployment improves responsiveness by reducing the distance that latency-critical data must travel and offloading work from the cloud to local nodes. At the same time, it increases sensitivity to network conditions and placement decisions: each network hop between microservices incurs latency and consumes bandwidth, and shared microservices often become hotspots as multiple application workflows converge on them. In the remainder of this paper, we use the term *upstream microservice* (UM) to denote the caller in a dependency pair and *downstream microservice* (DM) to denote the callee. Scheduling decisions determine where microservice pods are placed on cluster nodes and under what conditions pods are migrated between nodes.

### B. Challenges

Figure 1a illustrates an envisioned workflow of microservices executing across multiple nodes under dynamic network conditions in a cloud–edge continuum. Building effective scheduling policies and rigorously evaluating them requires addressing at least three major challenges.

*1) Dynamic Networking Conditions:* In cloud–edge scenarios, microservice applications often experience dynamic, heterogeneous networking conditions. Cross-node delays and available bandwidth can vary due to contention, background traffic, and routing changes. As a result, fluctuating cross-node latencies and bandwidth constraints significantly affect end-to-end response times and throughput. Evaluating how scheduling policies behave under such conditions requires a framework that can emulate and measure realistic, time-varying network profiles, rather than relying solely on the low-latency conditions of a laboratory cluster.

*2) Variability in Call Graph Topologies:* Microservice requests traverse complex call graphs, which vary depending on the types of incoming requests. As shown in Figure 1b, widely used benchmark applications such as Social Network [27] expose different request types (e.g., compose-post, read-home-timeline, read-user-timeline), each triggering a distinct call graph structure. The relative proportions of these request types change over time, leading to evolving mixtures of call graphs and shifting bottleneck edges. Scheduling policies must therefore handle both topology variability (which services are involved) and workload variability (how frequently each path is exercised).

*3) Imbalanced Traffic Distribution:* Even within a fixed call graph topology, traffic loads across UM–DM pairs can be highly imbalanced. Some service pairs carry orders of magnitude more requests or larger payloads than others, and this imbalance becomes more pronounced under high load. Figure 1c shows an example from the Social Network application where a small subset of UM–DM pairs accounts for the majority of traffic at 5k QPS. The pair `user-timeline-service` $\rightarrow$ `user-timeline-mongoDB`, for instance, experiences much higher traffic than other edges in the same call graph. Such imbalances can cause localized bottlenecks that dominate end-to-end latency.

These challenges have distinct causal roles. During monitoring interval $I_t$, let $\mathcal{M}_t$ denote the set of running microservices in the evaluated application namespace, and let $\boldsymbol{\lambda}_t$ denote the arrival rates of the root-request types. Executing this input through the deployed application triggers the active weighted call graph $G_t = (\mathcal{M}_t, E_t, w_t)$: $E_t$ contains the UM–DM pairs observed during $I_t$, and $w_t$ records their traffic stress. Independently, the network profile contains the directed node-pair delay and bandwidth matrices $D_t$ and $B_t$. Thus, $\boldsymbol{\lambda}_t$, $D_t$, and $B_t$ are configured experimental inputs; $\mathcal{M}_t$, $E_t$, and $w_t$ constitute the realized application state; and the service-to-node mapping $\pi_t$ is the policy output.

Joint control is necessary because service placement couples application-level communication edges with node-pair communication paths. Equation 5 formalizes this coupling: the request mix determines which edges belong to $E_t$ and their corresponding weights $w_t$, while $\pi_t$ maps those edges onto node pairs whose communication costs depend on $D_t$. Bandwidth-intensive transfers may also be constrained by $B_t$. For example, suppose request type A makes edge $e_A$ dominant and type B makes $e_B$ dominant. A placement that co-locates the endpoints of $e_A$ but maps $e_B$ across a degraded link may outperform an alternative placement when type A dominates and underperform it when type B dominates, even under the same aggregate QPS. A workload-only experiment would miss the path degradation, whereas a network-only experiment with a fixed call graph would miss the shift in communication hotspots. Controlling both factors within a unified workflow therefore supports isolated interventions for causal attribution and combined interventions for exposing their interactions.

## IV. iDynamics Framework

Motivated by the challenges discussed above, we propose `iDynamics`, a unified emulation framework for implementing and evaluating microservice scheduling policies in cloud–edge Kubernetes clusters. Specifically, `iDynamics` is designed to: (i) run real containerized microservices and workloads; (ii) expose call graph, traffic, and network conditions as controllable experimental factors; and (iii) provide a pluggable interface where different scheduling policies can be implemented, executed, and compared. Figure 2 presents the architecture of `iDynamics` and the interactions among its components during evaluation and scheduling. The framework supports managing diverse dynamics and implementing scheduling policies across the cloud–edge continuum. It currently targets Kubernetes and Istio, but the design principles generalize to other orchestrators and service meshes.

The key components of `iDynamics` are:

- **Graph Dynamics Analyzer**: This component consists of a UM–DM Traffic Profiler and a Call-Graph Builder. It reconstructs the call graph topology triggered by different request types and quantifies bi-directional traffic between microservices (and their replicas) under varying workloads (Section V).
- **Networking Dynamics Manager**: This component includes an Emulator and a distributed Measurer. The Emulator generates configurable cross-node latency and bandwidth patterns using Linux traffic control, while the Measurer collects actual communication metrics between nodes via lightweight daemon agents deployed across the cloud–edge cluster (Section VI).
- **Scheduling Policy Extender**: This component provides a Policy Customization Interface and a Utility Function Module. It enables rapid development and evaluation of scheduling strategies by exposing a policy-agnostic abstraction of nodes, pods, and metrics, and by providing helper functions to access cluster and network state (Section VII).

As depicted in Figure 2, the working procedure of `iDynamics` consists of multiple stages:

① Users submit realistic or synthetic workloads (with different request types and queries per second) to the deployed microservice applications.

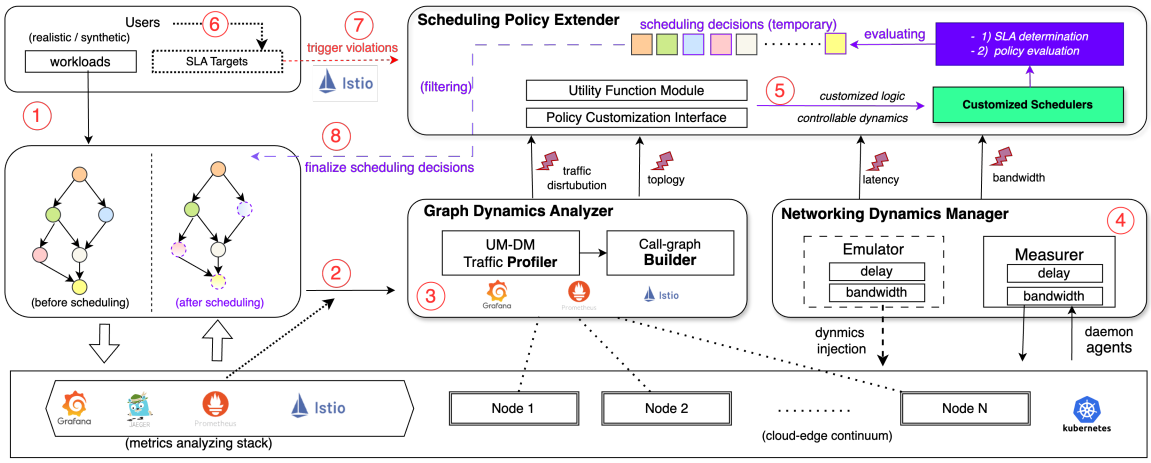



Fig. 2. `iDynamics` framework architecture and working procedure.

②-③ Performance metrics are collected using a monitoring stack (e.g., Istio [26], Jaeger [30], Prometheus [31]). The Graph Dynamics Analyzer then builds the triggered call graph and analyzes traffic between microservices, forwarding these insights to the Scheduling Policy Extender.

④ In parallel, the Networking Dynamics Manager measures real-time cross-node conditions and can optionally inject dynamic delay and bandwidth patterns into the cloud–edge environment.

⑤ Using the collected dynamics (call graphs, traffic distributions, and node-level latency and bandwidth), users implement and evaluate customized scheduling policies via the policy interface and utility module. In this context, a *scheduling decision* is a mapping of one or more pods to target nodes, possibly involving migrations to new nodes.

⑥ Users specify SLA targets (e.g., tail latency thresholds) that determine when a policy should be triggered.

⑦ A scheduling policy can be triggered either by tightening SLA criteria, which shifts performance classifications between *compliance* and *violation*, or by injecting more intense dynamics (e.g., increased delay or workload), which can cause actual performance degradation. Candidate scheduling decisions produced by the policy are stored in a decision queue and can be filtered based on compliance requirements or constraints on which services are allowed to migrate.

⑧ Finally, the filtered scheduling decisions are executed by reconfiguring pod placements in the Kubernetes cluster, adjusting the deployment according to current conditions.

With `iDynamics`, scheduling policies can thus be implemented, tested, and refined under realistic yet controllable conditions. For example, traffic- and latency-aware scheduling strategies can be developed using dynamic metrics from the Graph Dynamics Analyzer, emulated network dynamics from the Networking Dynamics Manager, and the extensible interfaces provided by the Scheduling Policy Extender.

## V. Graph Dynamics Analyzer

Considering the characteristics of microservice call graphs described in Section III, the *Graph Dynamics Analyzer* is designed to analyze (i) topology dynamics, i.e., variations in triggered call graphs, and (ii) traffic dynamics, i.e., imbalanced traffic distribution, of microservice applications deployed in the cloud–edge continuum. To capture these dynamics, `iDynamics` leverages a service mesh and implements two core modules: the *UM–DM Traffic Profiler* and the *Call-graph Builder*, as illustrated in Figure 2.

### A. Service Mesh

A service mesh enables monitoring of service-to-service traffic across microservice instances. A key requirement of the *Graph Dynamics Analyzer* is to capture and aggregate bidirectional traffic between dependent microservices efficiently. When there is only a single instance of each microservice, low-level traffic statistics can be collected from the node operating system. However, modern microservice applications typically scale individual services horizontally. Multiple upstream microservice (UM) and downstream microservice (DM) replicas introduce many-to-many communication patterns, making naive per-instance traffic analysis expensive and fragile. To address this, `iDynamics` adopts a service-mesh-based approach that offloads traffic collection and aggregation to sidecar proxies.

Although the current system uses Istio, the Graph Dynamics Analyzer only requires telemetry records that expose source workload, destination workload, and traffic-volume counters. Other service meshes or observability stacks can therefore be supported by exporting the same UM–DM traffic schema.

*1) Istio Service-Mesh Implementation:* To obtain fine-grained bidirectional traffic metrics between UM and DM microservices and their replicas, we implement the *UM–DM Traffic Profiler* using the Istio service mesh [26]. Istio is deployed as an infrastructure layer that transparently intercepts service-to-service traffic via sidecar proxies. In a Kubernetes cluster with Istio enabled, each microservice pod contains the application container and an Envoy sidecar container.

TABLE II
ISTIO SIDECAR OVERHEAD IN A CONTROLLED FORTIO BENCHMARK.

| Cluster scale | Config. | Thr. (rps) | ΔThr. (rps) | p95 (ms) | Δp95 (ms) | p99 (ms) | CPU (cores) | Mem. (MiB) |
|---|---|---|---|---|---|---|---|---|
| 5 | No-SC | 99.988 | – | 1.825 | – | 2.308 | 0.000 | 0.0 |
| 5 | Istio-SC | 99.961 | -0.027 | 6.123 | +4.299 | 6.971 | 0.033 | 66.9 |
| 20 | No-SC | 99.985 | – | 1.876 | – | 2.188 | 0.000 | 0.0 |
| 20 | Istio-SC | 99.953 | -0.032 | 6.944 | +5.069 | 8.631 | 0.039 | 70.0 |
| 45 | No-SC | 99.987 | – | 1.880 | – | 2.411 | 0.000 | 0.0 |
| 45 | Istio-SC | 99.955 | -0.032 | 7.021 | +5.141 | 7.613 | 0.043 | 68.8 |

*Notes:* No-SC and Istio-SC denote no-sidecar and Istio-sidecar configurations, respectively. Values are means over five valid repetitions using Fortio 1.69.3 with 100 QPS, 16 connections, a duration of 15 s, and 1024-byte payloads. Deltas are relative to the corresponding No-SC condition at the same worker-only placement-pool size. The CPU and memory columns report the additional combined client/server Istio sidecar footprint measured using Prometheus.

*2) Overhead Analysis:* Introducing a service mesh adds an additional proxy layer on the request path. We therefore measured its cost directly using a controlled two-service Fortio benchmark [32]. Table II reports the measured overhead of Istio sidecar injection. Across the three cluster scales, the no-sidecar configuration achieved p95 latencies of 1.825–1.880 ms, while the sidecar-enabled configuration increased p95 latency to 6.123–7.021 ms. This corresponds to an additional latency of 4.299–5.141 ms per request. Despite the additional proxy hop, throughput remained effectively unchanged at approximately 100 rps, with throughput reductions below 0.04 rps in all cases. Prometheus and cAdvisor (Container Advisor) monitoring further showed that the combined client/server sidecars consumed 0.033–0.043 CPU cores and 66.9–70.0 MiB of memory. These results indicate that Istio introduces a small but measurable latency penalty and modest resource overhead in our experimental environment.

### B. Call-graph Builder

The *Graph Dynamics Analyzer* periodically constructs an *active runtime call graph* for each application namespace. The graph is active because it is derived from traffic observed during a monitoring interval, rather than from static source-code dependencies. This distinction is important for microservice systems: production trace studies show that microservice call graphs are heavy-tailed, often tree-like, contain hotspot services, and can change topology across different request types or runtime contexts [3]. Accordingly, `iDynamics` treats call-graph topology and traffic stress as runtime experimental factors for evaluating scheduling policies under SLA targets.

Let $I_t = (t-\Delta t, t]$ denote a monitoring interval and let $\mathcal{M}_t$ be the set of running microservices in the namespace during that interval, with $M_t = |\mathcal{M}_t|$. For an ordered upstream–downstream service pair $(\mu, \sigma)$, where $\mu, \sigma \in \mathcal{M}_t$ and $\mu \neq \sigma$, the service mesh provides traffic counters grouped by source and destination labels. In this component implementation, $\mu$ denotes the upstream microservice (UM), $\sigma$ denotes the downstream microservice (DM), and the edge direction is determined by the source–destination label pair. The sent and received byte counters are used to quantify edge weight; they do not create a reverse edge unless the reverse source–destination pair is also observed.

**Algorithm 1** Build call graph with dependency topology and associated traffic stress.

**Input:** Application namespace $ns$, time interval $\Delta t$
**Output:** Directed sparse call graph $G$ with weighted traffic edges

Initialize $G$ as an empty directed graph
$MS \leftarrow \text{GETRUNNINGMS}(ns)$ ▷ all running microservices in $ns$
$(R, S) \leftarrow \text{QUERYMESHTRAFFICVECTORS}(ns, \Delta t)$ ▷ received/sent bytes grouped by source and destination labels
**for all** observed label pairs $(src, dst) \in \text{keys}(R) \cup \text{keys}(S)$ **do**
  **if** $(src \in MS)$ and $(dst \in MS)$ and $(src \neq dst)$ **then**
    $r \leftarrow \text{GETORZERO}(R, src, dst)$
    $s \leftarrow \text{GETORZERO}(S, src, dst)$
    $traffic \leftarrow (r + s)/(2\Delta t)$
    **if** $traffic > 0$ **then**
      $\text{ADDEDGE}(G, src, dst, traffic)$
    **end if**
  **end if**
**end for**
**return** $G$

**Stress Element.** A Stress Element (SE) models the observed runtime interaction between an upstream microservice and a downstream microservice. Let $B^{\text{sent}}_{\mu,\sigma}(I_t)$ and $B^{\text{recv}}_{\mu,\sigma}(I_t)$ denote the bytes sent and received, respectively, for the labelled pair $(\mu, \sigma)$ during $I_t$. We define the traffic stress of this pair as the mean bidirectional byte rate:

$$\text{stress}_t(\mu^{UM}, \sigma^{DM}) = \frac{B^{\text{sent}}_{\mu,\sigma}(I_t) + B^{\text{recv}}_{\mu,\sigma}(I_t)}{2\Delta t}. \quad (1)$$

In `iDynamics`, these quantities are computed from Prometheus metrics such as istio_tcp_sent_bytes_total and istio_tcp_received_bytes_total, aggregated over source workload and destination workload labels. A Stress Element is represented as

$$SE_t(\mu^{UM}, \sigma^{DM}) = (\mu, \sigma, \text{stress}_t(\mu, \sigma)).$$

The active edge set during $I_t$ is therefore

$$E_t = \{(\mu, \sigma) \mid \mu, \sigma \in \mathcal{M}_t,\ \mu \neq \sigma,\ \text{stress}_t(\mu, \sigma) > 0\}. \quad (2)$$

The Call-graph Builder returns a directed weighted graph

$$G_t = (\mathcal{M}_t, E_t, w_t),$$

where $w_t(\mu, \sigma) = \text{stress}_t(\mu, \sigma)$. Thus, vertices correspond to microservices and edges correspond to observed non-zero UM–DM communication during the interval. The absence of an edge means that no service-mesh traffic was observed for the pair during $I_t$; it does not rule out an inactive static code-level dependency.

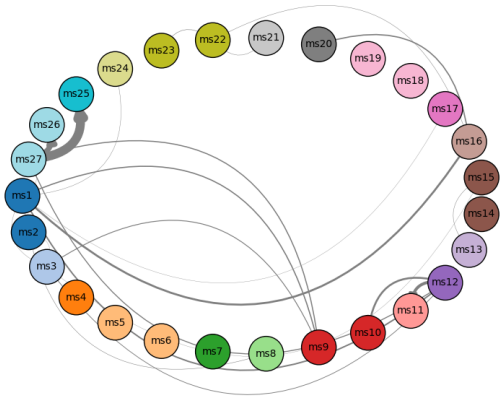



Fig. 3. The Graph Dynamics Analyzer uses Algorithm 1 to construct an active call-graph snapshot for an application with 27 microservices. The graph shows observed dependencies and traffic flows.

A dense all-pairs telemetry call graph builder enumerates the full $M_t(M_t - 1)$ adjacency space even though most candidate pairs may have zero traffic. By contrast, Algorithm 1 constructs the active graph in a sparse way, which directly aggregates service-mesh traffic vectors. It first retrieves sent and received traffic vectors for the namespace and interval, and then materializes only observed non-zero source–destination label pairs. This design is exact with respect to the monitoring interval: every positive traffic pair returned by the aggregate telemetry vectors is materialized as an edge, while zero-traffic pairs are not explicitly stored. For $M_t$ services and $|E_t|$ active edges, the *dense* all-pairs construction approach would perform $O(M_t^2)$ pair checks and, in our measured implementation, issues $2M_t(M_t - 1)$ pairwise Prometheus queries. In contrast, the *sparse* approach uses only two aggregate Prometheus queries and $O(|E_t|)$ edge materialization after the aggregate vectors are returned. If a workload genuinely activates most service pairs, then $|E_t|$ approaches $M_t(M_t-1)$ and materialization becomes quadratic; however, the telemetry query load remains fixed at two aggregate Prometheus queries.

For each monitoring interval, the constructed call graph captures both topological dynamics and traffic dynamics. In particular, traffic-aware policies can colocate high-traffic UM–DM pairs or place their pods on nodes connected by low-latency links, while hybrid policies can combine call-graph stress with the network delay and bandwidth state exposed by the Networking Dynamics Manager. Figure 3 illustrates a call graph snapshot for a 27-service Social Network [27] and provides an explanatory visualization of the reconstructed dependencies and traffic flows; it is not used as quantitative evidence. **As $G_t$ is reconstructed periodically, overlapping request mixes are represented naturally**: the active graph for an interval contains the union of the UM–DM pairs exercised during that interval, and their weights reflect the aggregate traffic stress contributed by those requests. Thus, `iDynamics` is not limited to hard switching between pre-defined static call graphs; it can also expose gradual changes in edge weights and active dependencies as the request mix evolves.

## VI. Networking Dynamics Manager

The *Networking Dynamics Manager* provides two core capabilities in `iDynamics`: (i) emulation of controllable cross-node networking conditions and (ii) accurate measurement of those conditions in the cloud–edge continuum. It consists of two main components: the **Emulator** and the **Measurer**. Together, they enable systematic evaluation of scheduling policies under diverse delay and bandwidth configurations without requiring intrusive changes to production environments.

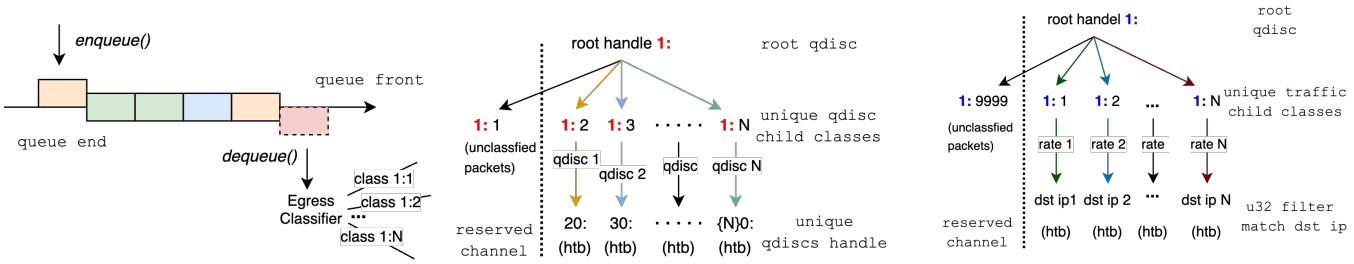
### A. Linux Traffic Control Primitives

Our emulation logic uses the Linux traffic control subsystem [33]. At the kernel level, we rely on queueing disciplines, classes, and packet classifiers; at the user level, we program these primitives together with the `tc` utility. Specifically, the implementation uses the following primitives: (i) **qdisc (queueing discipline):** Defines how packets are enqueued and dequeued on a network interface; (ii) **class:** A logical sub-queue within a classful qdisc, and each class can have its own rate and ceiling parameters and can host a child qdisc; (iii) **netem qdisc:** A classless queueing discipline used to emulate network impairments such as delay, jitter, loss, and reordering; (iv) **u32 filter:** A general-purpose packet classifier that matches fields in the packet header (e.g., source/destination IP address, ports, protocol) by applying mask-and-shift operations on 32-bit words; (v) **`tc` utility (traffic control):** A user-space utility from the `iproute2` suite that configures the kernel traffic control subsystem. It installs, updates, and removes qdiscs, traffic classes, and filters on network interfaces, enabling emulation of bandwidth caps, delay, jitter, or loss.

In `iDynamics`, we combine a classful `htb` (Hierarchical Token Bucket) root qdisc, per-destination classes with optional `netem` children, and `u32` filters to (i) inject configurable cross-node delays and (ii) shape available bandwidth per destination, while keeping non-experimental traffic (e.g., traffic to the control plane or the public Internet) unaffected.

### B. Emulator: Destination-oriented Design

*1) Emulation of Customized Delays:* Emulating diverse cross-node delays among cluster nodes is crucial for evaluating the robustness of various scheduling policies in the cloud-edge continuum. However, implementing customized communication delays from a single source node to multiple destination nodes poses significant challenges. To the best of our knowledge, as discussed in related work [5], [22]–[25], none of the existing studies has proposed an efficient method (i.e., one that is both simple and rapid) for injecting tailored communication delays in a controllable manner. In practical cloud-edge continuum environments, these approaches generally exhibit two main limitations: (1) the use of uniform communication delays from a single source node to all destination nodes prevents differentiation between node pairs, as delays from the source to all other nodes are identical; and (2) other networking services that do not involve the correlated nodes suffer degradation because the injected delays affect all outgoing traffic.




(a) Packet egress with classful disciplines. (b) Classful `qdisc` for delay emulation. (c) Classful `qdisc` for bandwidth shaping.

Fig. 4. The proposed method of destination-oriented networking emulation for (a) Packets are enqueued and dequeued according to classful disciplines. (b) Destination-specific delay emulation. (c) Destination-specific bandwidth shaping.

To address these limitations, we propose a destination-oriented emulation method that classifies packets using filters to distinguish egress packets and directs them based on their IP destinations (see Figure 4). Additionally, we reserve an extra channel for default packet transmission without injected delays, ensuring that the performance of other services remains unaffected, for example, packets to or from the control-plane node and external Internet destinations such as `google.com`. We implemented this scheme using the traffic control primitives of qdiscs (queuing disciplines) and the htb in Linux. The packet egress structure in Figure 4a shows where the per-destination classes are attached before the delay and bandwidth subclasses in Figures 4b and 4c. To emulate cross-node delays more realistically, we incorporate several factors: the base latency *bl* (representing the ideal minimal latency); the maximum additional latency *mal*; the distance factor *df* (accounting for latency due to physical separation); and the congestion factor *cf* (which simulates network uncertainties induced by traffic congestion). Here, $\textsc{RandUni}(0, mal)$ denotes the additional delay generated according to a random uniform distribution between 0 and $mal$. Thus, the emulated communication delay from node $i$ to node $j$ in a cluster of $N$ cloud–edge nodes is given by the following equation:

$$\text{delay}_i^j = \left\lfloor \left( bl + \textsc{RandUni}(0, mal) \times \frac{|i-j|}{N} \right) \times cf \right\rfloor \quad (3)$$

*2) Emulation of Customized Bandwidths:* Shaping customized available bandwidths between different cloud–edge node pairs is also crucial for creating dynamic networking conditions, thereby enhancing the generalizability of evaluated scheduling policies. Prior research—such as the work presented in [11] and [10]—has demonstrated the feasibility of emulating dynamic network conditions, including bandwidth variations, using decentralized emulation techniques. However, these approaches typically use a global or node-level configuration that does not support fine-grained per-destination control. In contrast, our proposed approach leverages traffic control primitives, including unique traffic classes and the u32 filter, to match different IP destinations. As illustrated in Figure 4c, this approach enables the emulation of distinct bandwidth settings for different destination nodes, even when originating from the same source cluster node. This fine-grained control is essential for accurately emulating heterogeneous network conditions in cloud–edge environments.

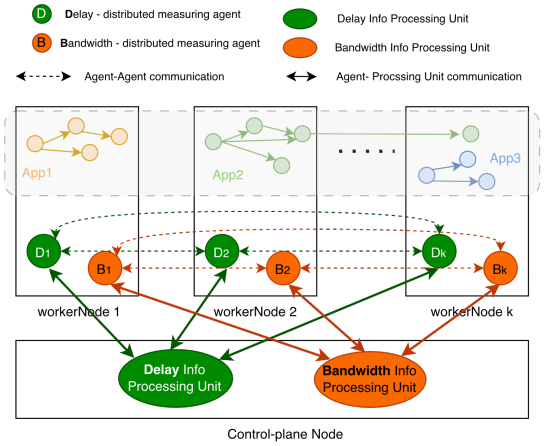



Fig. 5. Design of *Measurer* for delay and bandwidth measurement through distributed agents across cloud–edge nodes.

## C. Measurer: Distributed Agent Design

In practical cloud–edge computing environments, fluctuating network conditions significantly impact microservice performance and SLA compliance, including varying delays and bandwidths between nodes. High latency between cluster nodes negatively affects microservice communication, while dynamically changing bandwidth can lead to congestion, increased packet loss, and potential SLA violations. To accurately capture these dynamic cross-node conditions, we introduce the **Measurer**, a distributed agent-based measurement module within the *Networking Dynamics Manager* component of `iDynamics` (Figure 2).

**Design Overview:** The **Measurer** uses a unified approach to capture cross-node delays and bandwidths via distributed measurement agents. Although delay and bandwidth measurements follow a similar structure, we illustrate the design using delay measurement for clarity. The module consists of a centralized information processing unit and distributed lightweight agents. The centralized processing unit maintains minimal connections with agents and efficiently aggregates the collected data. The distributed agents run as lightweight containers deployed across cluster nodes, dedicated to measuring

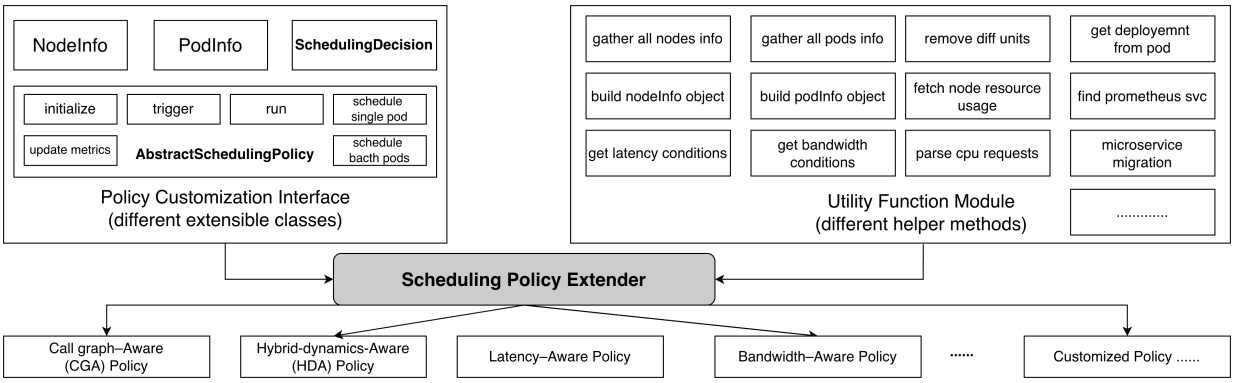



Fig. 6. Overview of the **Scheduling Policy Extender**, highlighting extensible classes, utility functions, and example policies.

and reporting network metrics. We implement an automatic scaling mechanism for these agents to ensure robustness and adaptability during cluster scaling. When new nodes join the cluster, corresponding agents are automatically instantiated, ensuring consistent and accurate measurements. Conversely, when nodes are removed, their associated agents are gracefully terminated.

**Implementation Details:** The centralized information processing unit is implemented as a plugin running on the control-plane node, collecting measured metrics from distributed agents via standard TCP communications. The distributed agents are deployed as Kubernetes DaemonSet pods, ensuring each node automatically hosts a dedicated measurement pod. These agents continuously measure and report cross-node delays and bandwidths, dynamically adjusting their presence in response to cluster scaling events, as depicted in Figure 5.

**Overhead Analysis:** To reduce measurement latency, the delay and bandwidth measurement tasks run concurrently. Each measurement result is aggregated into a result dictionary. Additionally, the distributed delay-measuring agents are lightweight and stable. Each node runs a delay-measuring agent built from the curlimages/curl image, consuming about 0.2 MiB of memory per node. Similarly, each node runs a bandwidth-measuring agent built from the networkstatic/iperf3 image, consuming around 0.84 MiB of memory per node. As the measurement tasks are designed to run in parallel, the measurement rate can be configured by changing the concurrency level, depending on the current load levels in the cluster.

## VII. Scheduling Policy Extender

The *Scheduling Policy Extender* provides the abstraction and tooling necessary to design, implement, and evaluate customized scheduling policies on top of `iDynamics`. By extending the provided interfaces and leveraging utility functions, users can efficiently develop tailored scheduling strategies that meet the diverse performance, scalability, and reliability requirements of microservice-based applications in cloud–edge environments.

### A. Policy Customization Interface

As illustrated in Figure 6, the policy interface is defined by the abstract class AbstractSchedulingPolicy, which establishes the fundamental structure for creating scheduling strategies. This abstract class includes predefined attributes and abstract methods essential for scheduling decisions. Scheduling decisions are encapsulated into combined NodeInfo and PodInfo objects. The NodeInfo class includes node-specific attributes, such as resource capacities (e.g., CPU and memory) and network characteristics (e.g., latency and bandwidth). The PodInfo class contains essential pod details, including resource requests, limits, and SLA requirements (e.g., throughput, response time), and is extensible to accommodate additional metrics. With this interface, researchers and cloud practitioners can implement their customized scheduling methods by overriding these logics, such as single-pod scheduling, batch-pod scheduling, and update-metrics methods triggered by adaptive scheduling signals. Additionally, Figure 7 makes the policy execution contract explicit. A newly constructed policy is in the `NEW` state. The `initialize` method validates policy configuration and advances it to `INITIALIZED`; `update_metrics` then installs one immutable, frame-aligned scheduling snapshot and advances it to `METRICS_READY`. The `trigger` method accepts only a signal for that same observation frame and advances the instance to `TRIGGERED`. If the signal does not request rescheduling, `run` returns no plan. Otherwise, `run` invokes the policy's batch-scheduling method, validates the returned decisions, caches the resulting batch, and advances to `PLANNED`. Supplying a newer runtime snapshot through `update_metrics` invalidates the previous trigger and cached batch and returns the policy to `METRICS_READY`; out-of-order transitions and mismatched frame identifiers are rejected.

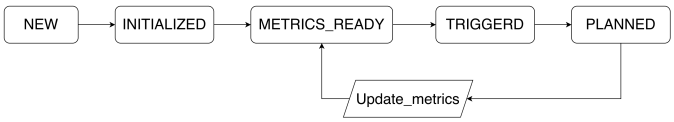



Fig. 7. Observable lifecycle states of one scheduling-policy instance.

### B. Utility Function Module

The utility function module complements the policy interface, providing ready-to-use functions that simplify and accelerate the extension of custom policies. At the node

level, it includes functions for collecting real-time node resource usage and network metrics, transforming them into structured objects compatible with scheduling algorithms. At the pod level, utility functions help manage pod resource specifications, SLA requirements, and metric unit conversions. Furthermore, additional helper functions utilize monitoring tools such as Prometheus to gather comprehensive cluster-level metrics and network conditions. By leveraging these pre-built utility functions, practitioners and researchers can rapidly and effectively develop sophisticated scheduling policies tailored to specific operational contexts.

### C. Examples of Customized Policies

To demonstrate the expressiveness of the policy interface, we implement the following reference policies for responding to cloud–edge dynamics:

**CGA (Call-graph–Aware) Policy:** This policy leverages the call graph and traffic stress information from the *Graph Dynamics Analyzer*. Microservice pairs with high traffic stress are preferentially co-located on the same node, subject to resource-capacity constraints. This reduces cross-node communication for hot dependencies and mitigates SLA violations caused by communication bottlenecks.

**HDA (Hybrid-dynamics–Aware) Policy:** This policy jointly considers call graph structure, traffic stress, and cross-node delays. It formulates microservice placement as a Service-Node Mapping Problem: microservices with high mutual traffic are mapped to nodes with low mutual delays to minimize total communication cost. This policy illustrates how multiple dynamics can be integrated within a single optimization formulation.

**Policy cost models.** To make the example policies more concrete, we briefly formalize their underlying objectives using the notation introduced in Sections V and VI. For monitoring interval $I_t$, the Graph Dynamics Analyzer provides the active weighted runtime call graph $G_t = (\mathcal{M}_t, E_t, w_t)$, where $\mathcal{M}_t$ is the set of running microservices, $E_t$ is the set of observed directed UM–DM edges, and $w_t(\mu, \sigma) = \mathrm{stress}_t(\mu, \sigma)$ is the traffic stress defined in Section V . Let $\mathcal{N}$ denote the set of worker nodes. The Networking Dynamics Manager provides the current cross-node delay $\mathrm{delay}_i^j$ from node $i$ to node $j$, following Equation 3. A placement for interval $I_t$ is a mapping $\pi_t : \mathcal{M}_t \to \mathcal{N}$ that assigns each microservice to a worker node.

For *CGA Policy*, we can define a simple affinity score that quantifies how preferable a candidate node $n \in \mathcal{N}$ is for placing a microservice $\mu$, given a tentative placement $\hat{\pi}_t$ of its communication partners:

$$\mathrm{score}_t^{\mathrm{CGA}}(\mu, n) = \sum_{\substack{(\mu,\sigma)\in E_t \\ \hat{\pi}_t(\sigma)=n}} w_t(\mu, \sigma) + \sum_{\substack{(\sigma,\mu)\in E_t \\ \hat{\pi}_t(\sigma)=n}} w_t(\sigma, \mu) \quad (4)$$

Intuitively, $\mathrm{score}_t^{\mathrm{CGA}}(\mu, n)$ is high when many of $\mu$'s heavily communicating upstream or downstream neighbors in $E_t$ are already placed on $n$. In our implementation, CGA Policy greedily assigns or migrates microservices to nodes that maximize $\mathrm{score}_t^{\mathrm{CGA}}(\mu, n)$, subject to per-node resource constraints (e.g., CPU and memory capacity), thereby co-locating hot dependencies and reducing cross-node traffic.

For *HDA Policy*, we introduce an explicit Service–Node Mapping objective that combines traffic stress and cross-node delays. The total communication cost of a placement $\pi_t$ during interval $I_t$ is defined as:

$$C_t(\pi_t) = \sum_{(\mu,\sigma)\in E_t} w_t(\mu, \sigma) \cdot \mathrm{delay}_{\pi_t(\mu)}^{\pi_t(\sigma)} \quad (5)$$

The ideal service placement then solves $\min_{\pi_t} C_t(\pi_t)$, subject to per-node resource-capacity constraints. In other words, active UM–DM edges with high runtime traffic stress are mapped to low-delay node pairs to minimize the aggregate communication cost along the current active call graph.

Note that these example policies are not meant to be exhaustive but show how `iDynamics` can support different strategies that leverage different aspects of the exposed dynamics.

## VIII. Performance Evaluation

This section evaluates whether `iDynamics` can construct, emulate, measure, and expose the dynamics required for microservice scheduling-policy evaluation. Rather than claiming that the example policies are universally optimal, our goal is to validate the framework mechanisms and demonstrate that they provide reproducible inputs for policy comparison. We organize the evaluation around the following research questions: (i) **RQ1:** What is the overhead of Algorithm 1 while building the active runtime call graphs? (ii) **RQ2:** How accurately and efficiently can `iDynamics` inject and measure cross-node latency and bandwidth dynamics? (iii) **RQ3:** Besides random matrices of networking states, can `iDynamics` replay trace-driven and burst-correlated network dynamics? (iv) **RQ4:** Can `iDynamics` represent continuous call-graph evolution under overlapping request mixes? (v) **RQ5:** Can `iDynamics` support different microservice-based applications and enable evaluation of call-graph-aware (CGA) and hybrid-dynamics-aware (HDA) policies?

### A. Cloud–Edge Testbed Setup

We implemented and evaluated `iDynamics` on a Kubernetes-based [4] cloud–edge testbed. The cluster contains a total of 46 virtual machine instance nodes, with one control-plane node and 45 worker nodes. The control-plane node has 16 CPU cores and 64 GiB of RAM. Among the worker nodes, 10 worker nodes have 4 CPU cores and 16 GiB of RAM, while the remaining 35 worker nodes have 2 CPU cores and 9 GiB of RAM. All worker and control-plane CPUs belong to the same AMD EPYC Milan processor family (`x86_64`). The cluster nodes communicate through a high-speed virtual network, with `iperf3` measurements showing approximately 16–22 Gbps of available bandwidth. The software stack uses Kubernetes v1.36.0, Calico v3.32.0 as the CNI plugin, Istio v1.30.0 as the service mesh, and CRI-O v1.36.0 as the container runtime. All nodes run Ubuntu 22.04.2 LTS with Linux kernel version 5.15.0.

The nodes are virtual machines hosted on the university's dedicated research cloud. The cloud resource manager controls the mappings of these VMs to physical compute hosts. The baseline inter-node latency is low (typically 0.2–1 ms in ICMP ping tests), so we do not treat the raw testbed as a naturally geo-distributed edge environment. Instead, `iDynamics` injects controlled cross-node delay and bandwidth constraints to emulate cloud–edge networking conditions.

### B. Overhead Analysis of Algorithm 1

To answer **RQ1**, we evaluate the overhead of Algorithm 1, which constructs the active runtime call graph used by the Graph Dynamics Analyzer (GDA). Table III reports both real benchmark deployments and synthetic service scales in the current cluster testbed. In the real Online Boutique [34] and Social Network [27] deployments, the call graph builder issues only two aggregate Prometheus queries instead of the logical dense baseline $2M(M-1)$, reducing telemetry query load by 132.0× and 702.0×, respectively. Their p95 total overheads are 11.405 ms and 10.354 ms. The local graph-build p95 is below 0.6 ms in both cases, so the live overhead is dominated by Prometheus aggregate-vector query latency.

The synthetic rows isolate local graph construction and query-count scaling at larger service counts. At 1,000 services with 4,000 active edges, sparse graph construction has a p95 build time of 42.300 ms, compared with 111.331 ms for the measured dense pair-scan loop and 1,998,000 logical dense queries. At 50,000 services with 200,000 active edges, the sparse builder still uses two telemetry queries and materializes the graph in 2696.727 ms, while the dense logical baseline would require 4,999,900,000 pairwise queries. Overall, these results show that Algorithm 1 bounds monitoring-plane overhead by replacing the dense $2M(M-1)$ pairwise query pattern with two aggregate telemetry queries. The remaining local build cost scales with the materialized sparse graph, i.e., the service/workload vertices and active directed edges, rather than all possible service pairs.

TABLE III
OVERHEAD OF ALGORITHM 1 ON REAL BENCHMARKS AND SYNTHETIC SERVICE SCALES.

| Case | Mode | Svc. | Edges | Queries | Q-red. | Query p95 (ms) | Build p95 (ms) | Total p95 (ms) |
|---|---|---|---|---|---|---|---|---|
| *Real benchmarks* | | | | | | | | |
| Online Boutique | sparse | 12 | 15 | 2 | 132.0× | 11.035 | 0.369 | 11.405 |
| Social Network | sparse | 27 | 24 | 2 | 702.0× | 9.897 | 0.532 | 10.354 |
| *Synthetic service scale* | | | | | | | | |
| 1,000 services | dense | 1K | 4K | 1,998K | 1.0× | – | 111.331 | – |
| 1,000 services | sparse | 1K | 4K | 2 | 999.00K× | – | 42.300 | – |
| 5,000 services | sparse | 5K | 20K | 2 | 25.00M× | – | 211.071 | – |
| 10,000 services | sparse | 10K | 40K | 2 | 99.99M× | – | 472.450 | – |
| 20,000 services | sparse | 20K | 80K | 2 | 399.98M× | – | 974.990 | – |
| 50,000 services | sparse | 50K | 200K | 2 | 2.50B× | – | 2696.727 | – |
| 50,000 services | dense | 50K | 200K | ∼5B | 1.0× | – | – | – |

*Notes:* All time columns report p95 latency in milliseconds. *Svc.* denotes the deployed service/workload vertices used by the call-graph builder; in real benchmark rows, it counts Kubernetes deployments with positive replicas in the evaluated namespace. For real benchmark rows, *Query* is the Prometheus aggregate-vector latency, *Build* is the local wall-clock time required to materialize the call graph from returned telemetry edges, and *Total* is their sum. For synthetic rows, *Query* and *Total* are not reported because no live Prometheus query is issued; *Build* reports sparse-graph materialization time for sparse rows and dense pair-scan time for the measured 1,000-service dense baseline. *Q-red.* is the query-count reduction relative to the logical dense pairwise baseline $2M(M-1)$. Dense synthetic timing is measured only at 1,000 services; larger dense rows are query-count baselines only.

TABLE IV
ACCURACY OF CROSS-NODE NETWORK EMULATION.

| Metric | Pairs | Mean | Median | p95 | Max |
|---|---|---|---|---|---|
| Delay abs. error (ms) | 72 | 0.274 | 0.155 | 0.852 | 1.210 |
| Bandwidth abs. error (Mbit/s) | 72 | 21.85 | 14.00 | 89.45 | 214.00 |
| Bandwidth rel. error (%) | 72 | 4.28 | 3.00 | 13.66 | 27.47 |

*Notes:* Statistics are computed over all directed worker-to-worker pairs in the 9-worker validation matrix, excluding self-pairs. Delay error is the absolute difference between injected and measured delay. Bandwidth error is the absolute or relative difference between saturated and measured bandwidth. Reserved channels were not delay-injected; measured delay remained 0.21–0.71 ms to the control-plane node and 14.8–15.3 ms to external `google.com`.

### C. Validation of Networking Dynamics Manager

To answer RQ2, we validate the design of *Networking Dynamics Manager*, which is responsible for injecting and measuring cross-node latency and bandwidth.

**Latency emulation accuracy:** We validate the destination-oriented delay-emulation scheme in Figure 4b by injecting distinct source–destination delays between worker nodes and measuring the resulting delays using the distributed measurement agents. As shown in Table IV, the mean absolute delay error is only 0.274 ms, with a median error of 0.155 ms and a p95 error of 0.852 ms. Even the maximum observed error is 1.210 ms. These results show that the measured delays closely track the injected values across heterogeneous node pairs. The remaining deviations are small and are expected in a virtualized Kubernetes environment due to OS scheduling, packet-processing variability, and measurement noise.

Importantly, the reserved channels are not affected by the injected worker-to-worker delays. Across the validation run, measured delays from worker nodes to the control-plane node remain between 0.21 and 0.71 ms, while delays to `google.com` remain between 14.8 and 15.3 ms. This confirms that the destination-oriented design can inject delays for selected worker-to-worker paths while preserving default communication channels for non-targeted traffic.

**Bandwidth shaping accuracy:** We similarly validate bandwidth shaping by saturating worker-to-worker paths using `iperf3`-based pods and comparing the measured throughput against the configured saturated bandwidth. Table IV shows that the mean absolute bandwidth error is 21.85 Mbit/s, with a median error of 14.00 Mbit/s. In relative terms, the mean error is 4.28% and the median error is 3.00%, indicating that the measured bandwidth generally follows the configured target closely. The p95 relative error is 13.66%, while the maximum relative error is 27.47%, showing that a small number of saturated paths exhibit larger deviations. This is expected for short-window bandwidth measurements in a shared virtualized testbed, where transient contention and TCP dynamics can affect saturated throughput. Overall, the results are sufficient for creating differentiated bandwidth regions, such as high-bandwidth and low-bandwidth node pairs, for

TABLE V
NETWORK TRACE GENERATION AND REPLAY STATISTICS.

| Latency | | | | Bandwidth | | | |
|---|---|---|---|---|---|---|---|
| **Metric** | **Dist.** | **Burst** | **CSV** | **Metric** | **Dist.** | **Burst** | **CSV** |
| p50 (ms) | 14.02 | 18.59 | 18.59 | p50 (Mbit/s) | 517.65 | 368.33 | 368.33 |
| p95 (ms) | 42.83 | 45.12 | 45.12 | p95 (Mbit/s) | 776.48 | 574.46 | 574.46 |
| p99 (ms) | 58.83 | 63.63 | 63.63 | p99 (Mbit/s) | 795.06 | 615.24 | 615.24 |
| CV | 0.692 | 0.543 | 0.543 | CV | 0.333 | 0.286 | 0.286 |
| Peak/med. | 5.38 | 3.87 | 3.87 | Peak/med. | 1.54 | 1.72 | 1.72 |
| Lag-1 | -0.056 | 0.946 | 0.946 | Lag-1 | -0.015 | 0.958 | 0.958 |
| Spatial | -0.024 | 0.067 | 0.067 | Spatial | -0.019 | 0.444 | 0.444 |
| Burst duration (s) | 6.37 | 27.95 | 27.95 | Burst dur. (s) | 6.17 | 14.23 | 14.23 |

*Notes:* Dist., Burst, and CSV denote **distance-random provider**, **burst-correlated provider**, and **CSV-replay provider**, respectively. Statistics are computed over all directed non-self node pairs and all time steps. CV denotes coefficient of variation; Lag-1 denotes mean temporal autocorrelation; Spatial denotes correlation across directed node pairs. CSV-replay provider uses the archived burst-correlated matrix sequence and reproduces the same processed metrics.

controlled scheduling experiments.

Overall, Table IV shows that `iDynamics` can accurately emulate and measure heterogeneous cross-node network conditions. The low delay error confirms the effectiveness of destination-specific latency injection, while the bandwidth results show that the framework can reproduce differentiated bandwidth conditions with acceptable measurement variability. These capabilities provide the foundation for evaluating networking-related scheduling policies under controlled and repeatable cloud–edge dynamics.

### *D. Trace-driven and Burst-correlated Network Dynamics*

To answer **RQ3**, we evaluate whether the *Networking Dynamics Manager* can expose time-varying network states beyond independent distance-random matrices (used by validations in subsection VIII-C). The distance-random provider (Equation 3) is useful as a controllable stress baseline, but independent random samples do not preserve two important properties of practical cloud–edge networks: **temporal persistence**, where congestion lasts for multiple measurement intervals, and **spatial correlation**, where multiple paths are affected by a shared bottleneck or routing event. We therefore extend the *Networking Dynamics Manager* with a `NetworkTraceProvider` abstraction that decouples network-state generation from live traffic-control application.

A provider emits a sequence of network-state frames $F_k = (L_k, B_k)$, where $L_k \in \mathbb{R}^{N\times N}$ is a directed one-way latency matrix and $B_k \in \mathbb{R}^{N\times N}$ is a directed bandwidth matrix for the $k$-th time step. Diagonal entries are ignored because self-communication is not shaped. The framework supports three providers: (i) a **distance-random provider**, which is retained as the synthetic baseline; (ii) a **CSV-replay provider**, which replays archived time-indexed $N \times N$ latency and bandwidth matrices; and (iii) a **burst-correlated provider**, which generates latency spikes and bandwidth drops with configurable temporal correlation, spatial correlation, burst probability, burst duration, and jitter.

Table V shows that the distance-random baseline has near-zero temporal autocorrelation for both latency and bandwidth, indicating that consecutive samples are largely independent. By contrast, the burst-correlated provider preserves strong temporal persistence: the lag-1 autocorrelation is 0.946 for latency and 0.958 for bandwidth. It also produces longer network events, increasing the mean latency-burst duration from 6.37 s to 27.95 s and the mean bandwidth-burst duration from 6.17 s to 14.23 s. The bandwidth trace further exhibits spatial correlation of 0.444, showing that bandwidth drops can affect multiple paths together. These properties make the generated network states more suitable for evaluating schedulers under persistent congestion and correlated bottlenecks than independent random matrices.

TABLE VI
EXTERNAL LATENCY CALIBRATION AGAINST REAL RIPE ATLAS TRACES.

| **Trace Data** | **p50** | **p95** | **p99** | **CV** | **Peak/med.** | **Lag-1** |
|---|---|---|---|---|---|---|
| RIPE Atlas ( real ) | 22.63 | 75.72 | 79.65 | 1.86 | 116.0 | 0.066 |
| Fitted generator ( fitting ) | 24.50 | 64.96 | 83.99 | 0.520 | 4.551 | 0.859 |

*Notes:* The RIPE Atlas row is computed directly from 2,756 valid RTT samples from eight probes over the 2026-06-02 to 2026-06-03 window. The fitted generator row is computed from a synthetic latency matrix sequence generated by **burst-correlated provider**. Its parameters are selected by grid-search to match the RIPE sample's median and tail latency scale. The fitted generator is therefore not a replay of RIPE packets or paths. The fitting calibration is latency-only because RIPE Atlas ping does not provide bandwidth measurements.

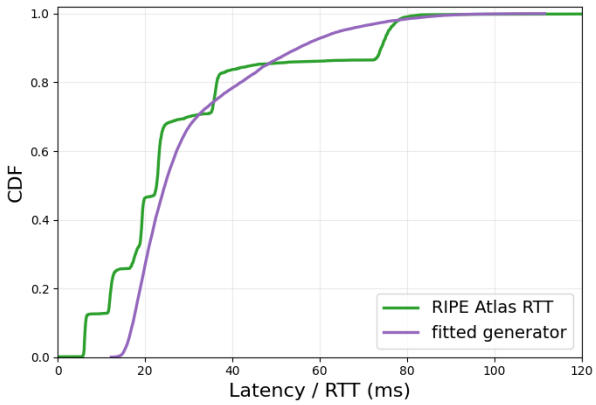



Fig. 8. Empirical CDF (Cumulative Distribution Function) of RIPE Atlas RTT samples and the fitted burst-correlated latency generator.

To connect the generated traces to an external public latency source, we further calibrated the burst-correlated latency generator against RIPE Atlas built-in IPv4 ping measurement 1001 toward `k.root-servers.net` [35]. RIPE Atlas is a global, open, co-operatively built internet measurement network designed to provide a real-time understanding of internet connectivity and reachability. In our evaluation, the calibration uses 2,756 valid round-trip time (RTT) samples from eight contributing probes over the specified window.

Table VI and Figure 8 show that the fitted generator matches the main latency scale of the public RTT sample: the median is 24.504 ms versus 22.626 ms in the RIPE sample, and the p99 is 83.998 ms versus 79.646 ms. However, the generator does not reproduce all properties of the public trace. In particular, the RIPE sample contains rare extreme outliers, reflected by a peak-to-median ratio of 116.0, whereas the fitted generator has a peak-to-median ratio of 4.551. The RIPE sample also

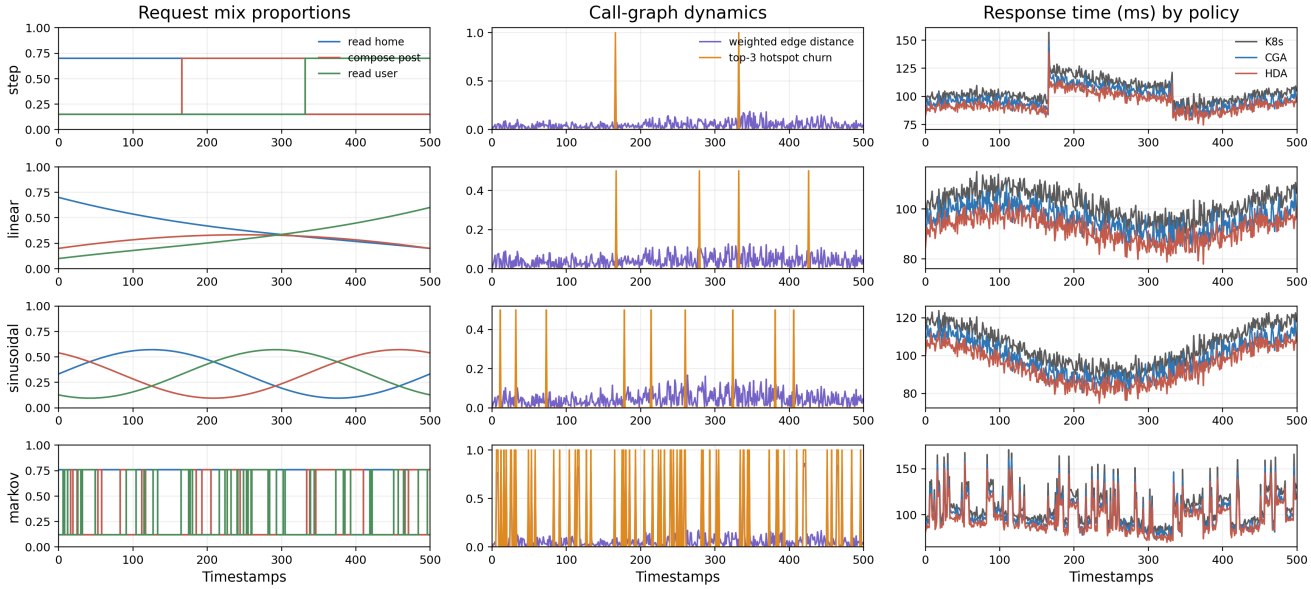



Fig. 9. Continuous call-graph evolution under varying request types, proportions, and mix modes for the Social Network benchmark. Columns display request-mix proportions, graph-dynamics metrics, and response times. Weighted edge distance quantifies normalized traffic-weight redistribution between adjacent call graph snapshots, while Top-3 hotspot churn reports whether the three dominant directed edges change in the graph. The evaluations are conducted at 90 QPS with measurements collected at 5-second intervals across the 45-worker node scale in the testbed.

has low lag-1 autocorrelation, while the selected generator intentionally preserves high temporal persistence. We therefore use this calibration to bound the latency scale of the generator, not to claim that one public RTT sample represents all wide-area network (WAN) or cloud–edge systems. Bandwidth remains synthetic in this calibration because the public ping measurement does not provide throughput data.

Overall, the results answer **RQ3**: `iDynamics` can support various networking dynamics by replaying archived time-indexed latency and bandwidth matrices, generating burst-correlated traces with temporal and spatial structure, and calibrating latency scale against the external public RTT traces.

TABLE VII
REQUEST-MIX GENERATION MODES.

| Mode | Semantics |
|---|---|
| `step` | One request class is dominant for each segment while all other classes remain active at baseline weight. |
| `linear` | Dominance shifts smoothly from the first request class to the last, with a midpoint bump for interior request classes. |
| `sinusoidal` | All request classes remain active while phase-shifted sinusoidal weights change the hot request over time. |
| `markov` | A Markov schedule changes the dominant request class stochastically while preserving nonzero baseline traffic for the rest. |

### *E. Continuous Call-graph Evolution and Policy Evaluation*

To answer **RQ4** and the policy-evaluation part of **RQ5**, we evaluate continuous call-graph evolution with mixed request types, proportions and modes. The workload mixer keeps multiple request classes active concurrently and changes their mix proportions under **step**, **linear**, **sinusoidal**, and **Markov** modes, summarized in Table VII. Consequently, each monitoring interval produces an active weighted call graph whose topology and edge weights reflect the combined effect of overlapping request classes. This setting exercises the core framework requirement that scheduling policies consume a stream of evolving graph and traffic loads, instead of a single static dependency graph.

We use two complementary call-graph metrics to separate the edge-weight movement (*traffic evolution*) from the changes of dominant active edges (*topology evolution*) in the call graph. Let $E_t$ be the active edge set of the directed weighted call graph at interval $t$, and let $w_t(e)$ be the nonnegative traffic-stress weight of edge $e$. The **weighted edge distance** is a normalized sum of absolute weight differences across all edges in the union of $E_{t-1}$ and $E_t$:

$$D_w(E_{t-1}, E_t) = \frac{\sum_{e \in E_{t-1} \cup E_t} |w_{t-1}(e) - w_t(e)|}{\sum_{e \in E_{t-1} \cup E_t} \max(w_{t-1}(e), w_t(e))}. \quad (6)$$

Missing edges are treated as weight zero. Thus, $D_w = 0$ means adjacent call graph snapshots have identical weighted traffic, whereas larger values mean that traffic has shifted, appeared, or disappeared across directed service pairs. The normalization makes the score comparable across different graph sizes and traffic levels.

**Top-3 hotspot churn** focuses on a different perspective: whether the main bottleneck edge candidates change in the graph over time. Let $H_t^{(3)}$ be the set of the three highest-weight directed edges in $E_t$. We compute

$$C_3(E_{t-1}, E_t) = 1 - \frac{|H_{t-1}^{(3)} \cap H_t^{(3)}|}{|H_{t-1}^{(3)} \cup H_t^{(3)}|}. \quad (7)$$

This is a Jaccard-distance score over the hotspot sets. The Jaccard distance is a metric used to quantify how dissimilar

TABLE VIII
PERFORMANCE OF CONTINUOUS REQUEST MIXES UNDER ROBUSTNESS EVALUATION.

| Steps | Mode | Entropy | Top-3 churn | Active edges | Response time (ms) | | |
|---|---|---|---|---|---|---|---|
| | | | | | K8s | CGA | HDA |
| 200 | Step | 1.181 | 0.010 | 13.3 | 104.52 | 98.25 | 93.83 |
| | Linear | 1.488 | 0.010 | 15.0 | 101.93 | 95.82 | 91.51 |
| | Sinusoidal | 1.387 | 0.023 | 13.3 | 104.78 | 98.50 | 94.07 |
| | Markov | 1.035 | 0.141 | 11.0 | 104.60 | 98.32 | 93.91 |
| 500 | Step | 1.181 | 0.004 | 13.3 | 104.38 | 98.14 | 94.01 |
| | Linear | 1.489 | 0.004 | 15.0 | 101.94 | 95.86 | 91.82 |
| | Sinusoidal | 1.387 | 0.009 | 13.3 | 104.73 | 98.47 | 94.32 |
| | Markov | 1.035 | 0.136 | 11.2 | 107.38 | 100.97 | 96.72 |
| 1000 | Step | 1.181 | 0.002 | 13.3 | 104.40 | 98.16 | 93.94 |
| | Linear | 1.489 | 0.002 | 15.0 | 102.02 | 95.93 | 91.80 |
| | Sinusoidal | 1.387 | 0.005 | 13.3 | 104.80 | 98.54 | 94.30 |
| | Markov | 1.035 | 0.155 | 11.5 | 109.75 | 103.20 | 98.76 |

*Notes:* Each metric reports the **mean value** across all evaluation timestamps in the corresponding horizon. Entropy is the Shannon entropy of the request-mix proportion vector, $H(P) = -\sum_i p_i \log_2 p_i$, with zero-probability terms omitted. Top-3 churn and active edges characterize temporal variation in the resulting call graph. K8s, CGA, and HDA report the measured performance under the corresponding scheduling policies; lower values are better.

two sets are. It measures the difference between sample sets by dividing the size of the set difference by the size of the set union. Scores range from 0 (identical) to 1 (completely disjoint). In Equation 7, a value of zero means the same three directed edges remain dominant; a high value means the heaviest traffic paths have moved to different service pairs.

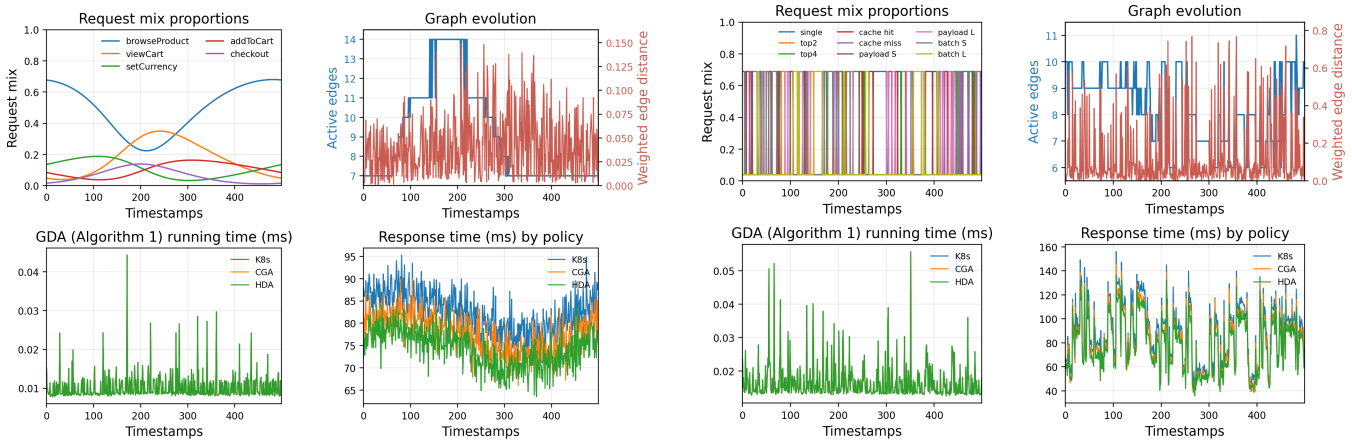
**Why use these two metrics?** An *active runtime call graph* can have low hotspot churn but nonzero weighted distance: the same top edges remain, but their traffic weights shift. The call graph can also have high hotspot churn while total traffic volume is similar: the bottleneck edges move to different service pairs. That distinction is useful for scheduling-policy evaluation. A scheduler may care about total traffic redistribution, but it may care even more when the dominant edges change in the call graph, because those are the graph edges most likely to affect microservice placement, colocation, delay sensitivity, or bandwidth pressure. Therefore, these two metrics are complementary in an *active runtime call graph*: weighted edge distance measures how much traffic load changes, while Top-3 hotspot churn measures whether the most policy-relevant hot edges change.

As shown in Figure 9, the qualitative differences among the four evolution modes are illustrated. Step mode produces piecewise-stable plateaus; linear and sinusoidal modes produce gradual shifts in the request-mix vector; and Markov mode produces bursty, correlated changes in the dominant request class. In Table VIII, the quantitative metrics show that linear mixes have the highest mean request-mix entropy and the largest active graph, with approximately 15 active directed edges. Markov mixes have lower entropy but much higher mean Top-3 hotspot churn, reaching 0.155 at the 1000-timestamp evaluation. This indicates a narrower but less stable hot-edge set, which is precisely the type of evolving bottleneck pattern that can stress microservice placement policies. Across all modes and evaluation horizons for the three evaluated policies, HDA has the lowest response time, followed by CGA and then K8s default. The important framework result is not that HDA is universally superior, but that the same evolving call graph can be replayed through multiple executable policy objectives over long horizons while preserving a clear evidence boundary.

### *F. Application Generality and Compatibility Case Studies*

To answer the application-generality part of **RQ5**, we evaluate whether `iDynamics` can preserve a common measurement and policy interface across applications with different request classes and application call graphs. We use two complementary benchmarks. Online Boutique [34] is an externally maintained e-commerce application whose gRPC service graph exercises conventional cloud microservice paths, such as browsing, cart, and checkout. It tests whether the framework can attach to a realistic benchmark that was not designed for `iDynamics`. The second benchmark is our CPU-only mixture of experts (MoE) serving microbenchmark, which deploys a role-selected HTTP service graph with a frontend, tokenizer, router/gate, cache, aggregator, and eight expert services. Its implemented request path is frontend $\rightarrow$ optional cache lookup $\rightarrow$ tokenizer $\rightarrow$ request-level top-$k$ router $\rightarrow$ selected experts $\rightarrow$ aggregator $\rightarrow$ cache update. We use it because MoE-style inference workloads introduce dynamic expert popularity, routing-dependent fan-out/fan-in traffic, cache effects, payload-size variation, and batch-size variation, which are qualitatively different from e-commerce (Online Boutique) request flows. Together, these benchmarks test adapter generality across an external application and a controllable ML-inference-style microservice graph; they are not intended to claim universal benchmark coverage, neural-network MoE inference fidelity, or full GPU-aware production large language model (LLM) serving.

Figure 11 summarizes the evaluated MoE service graph, and Figure 10 shows that the same request-mix, call-graph, GDA-runtime, and policy-specific response times can be measured for both benchmarks by `iDynamics`. In the Online Boutique panel (Figure 10a), the sinusoidal mix changes request dominance smoothly, and the reconstructed runtime graph changes accordingly: the active-edge count moves between smaller and larger active graphs while weighted-edge distance captures the traffic redistribution between call graph snapshots. The response time curves from different policies show a consistent ordering in this case study, with HDA generally producing lower values than CGA and the K8s default, while the Algorithm 1 construction cost remains small relative to the policy-level response-time scale. In the MoE panel (Figure 10b), the Markov request mix causes abrupt shifts among single-expert, multi-expert, cache, payload, and batch classes; this produces sharper graph-distance bursts and larger response-time variation because the hot expert and support-service paths move more abruptly. The evaluations suggest that `iDynamics` is not tied to a small set of benchmark-specific graphs (e.g., Social Network and Online Boutique): the same dynamics adapter scheme can expose gradual e-commerce workflow evolution and bursty expert-routing evolution through comparable metrics and executable scheduling policies, while keeping the MoE microbenchmark claim bounded by CPU-only service-graph and placement dynamics.




(a) Online Boutique under a sinusoidal request-mix schedule with 45 worker nodes and 5 replicas per microservice.

(b) CPU-only MoE-style microbenchmark under a Markov request-mix schedule with 45 worker nodes and 5 replicas per microservice.

Fig. 10. Application-generality examples for Online Boutique and the CPU-only MoE-style microbenchmark. Both panels use the same request-mix, weighted call-graph, policy-evaluation, and runtime-GDA schema.

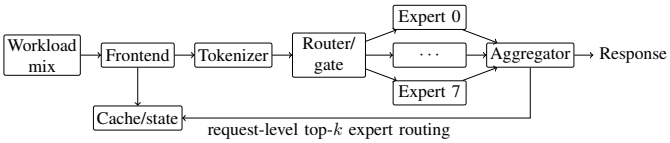



Fig. 11. CPU-only MoE-style serving microbenchmark used in the application-generality study. The benchmark exposes dynamic expert popularity, fan-out/fan-in service traffic, cache behavior, and payload effects for placement evaluation; it is not a faithful neural-network MoE inference engine.

### G. Threats to Validity and Limitations

The evaluation is conducted on a controlled Kubernetes research testbed hosted on virtual machine instances. The 45-worker node results demonstrate scalability and policy behavior in our current cluster, but they should not be interpreted as universal production performance for geographically distributed bare-metal edge deployments. VM execution may be affected by hypervisor scheduling, vCPU contention, co-location, noisy neighbors, and virtual-switch contention; because the mapping of VMs to PMs (physical machines) is unavailable, these effects cannot be isolated from application-level variance. Network dynamics are emulated with Linux `tc` and `qdisc` on a Calico overlay. Although each run records `qdisc` state, resets the configuration, and verifies connectivity, destination-worker shaping can affect other selected-worker overlay packets. We therefore treat the fault-injection results as controlled policy experiments, not as a proof of perfect traffic isolation.

The measurement stack also has limitations. Service-mesh telemetry provides the UM–DM traffic labels required by GDA, but it introduces sidecar overhead that depends on mesh configuration, payload size, request path, and security settings. We quantify this overhead in our environment, rather than claiming a mesh-independent cost. Similarly, runtime-GDA values in our evaluations measure the graph-construction path for archived snapshots; they represent live Prometheus overhead only when the corresponding experiment contains live query samples. The graph metrics used in the paper are compact summaries: weighted edge distance captures traffic redistribution, and Top-3 hotspot churn captures dominant-hotspot changes over time, but neither metric by itself explains the causal source of a performance change.

Finally, the benchmark and policy claims are intentionally bounded. Online Boutique supports external-application compatibility and replay-schema evidence, while the MoE case study is a CPU-only microservice benchmark for expert-skew, fan-out/fan-in, cache, payload, and placement dynamics. It does not model GPU topology, tensor parallelism, KV-cache locality, accelerator memory pressure, token-level MoE routing, or production LLM-serving schedulers. CGA and HDA are executable reference policies used to validate the Scheduling Policy Extender; they are not claimed to be state-of-the-art or universally optimal schedulers. Stronger objectives involving CPU, memory, energy, fairness, isolation, or accelerator resources can be implemented through the same interface, but require separate empirical validation.

## IX. Conclusions and Future Work

We proposed `iDynamics`, a configurable emulation framework for designing and evaluating microservice scheduling policies under dynamics in cloud–edge computing environments. Unlike prior work that typically focuses on specific scheduling algorithms or static testbeds, `iDynamics` offers an end-to-end experimentation environment. Extensive evaluations show that `iDynamics` can construct active runtime call graphs efficiently, validate live `tc`-based network impairments, replay continuous request-mix and call-graph evolution over long horizons, and compare representative scheduling policies on worker pools of up to 45 worker nodes. Additional Online Boutique and CPU-only MoE-style case studies show that the same adapter scheme can be applied beyond the Social Network benchmark, while keeping claims bounded to the evidence provided by each evaluation. Overall, the results show that `iDynamics` provides a repeatable and extensible

environment for studying how scheduling policies respond to workload, application graph, and network dynamics.

Future work will extend `iDynamics` in three directions. First, we will incorporate richer policy objectives and metrics, including CPU, memory, energy, fairness, isolation, and multi-tenant contention. Second, we will broaden benchmark coverage with a more realistic MoE implementation, stable event-streaming systems, and ML-serving workloads, including GPU-aware telemetry when suitable hardware is available. Third, we will investigate lower-overhead data-plane mechanisms, such as eBPF/XDP or traffic-control offload, to support richer network behaviors at larger scales.

**Software**: `iDynamics` framework prototype as open source is available at https://github.com/Cloudslab/iDynamics

## REFERENCES


[1] S. Pallewatta, V. Kostakos, and R. Buyya, "Placement of microservices-based iot applications in fog computing: A taxonomy and future directions," *ACM Computing Surveys*, vol. 55, no. 14s, July 2023.

[2] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with sla guarantees," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'23. New York, USA: ACM, 2022, p. 62–77.

[3] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*. New York, USA: ACM, 2021, p. 412–426.

[4] Kubernetes Authors, "Kubernetes: An open-source container orchestration system," https://kubernetes.io/, 2025, accessed: 2025-11-02.

[5] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 805–825.

[6] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: ACM, 2019.

[7] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: ACM, 2021, p. 135–151.

[8] H. Tian, Y. Zheng, and W. Wang, "Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: ACM, 2019, p. 139–151.

[9] A. Mirhosseini, S. Elnikety, and T. F. Wenisch, "Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: ACM, 2021, p. 442–457.

[10] L. Liechti, P. Gouveia, J. Neves, P. Kropf, M. Matos, and V. Schiavoni, "Thunderstorm: A tool to evaluate dynamic network topologies on distributed systems," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 241–250.

[11] P. Gouveia, J. a. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, "Kollaps: Decentralized and dynamic topology emulation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, USA: ACM, 2020.

[12] S. Schmid, C. Avin, C. Scheideler, M. Borokhovich, B. Haeupler, and Z. Lotker, "Splaynet: towards locally self-adjusting networks," *IEEE/ACM Trans. Netw.*, vol. 24, no. 3, p. 1421–1433, Jun. 2016.

[13] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. IEEE Press, 2021, p. 1–9.

[14] A. Marchese and O. Tomarchio, "Network-aware container placement in cloud-edge kubernetes clusters," in *Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 859–865.

[15] Marchese, Angelo and Tomarchio, Orazio, "Extending the kubernetes platform with network-aware scheduling capabilities," in *Proceedings of the 20th International Conference on Service-Oriented Computing (ICSOC 2022)*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 465–480.

[16] X. Zhu, Y. Li, L. Yao, Z. Qi, Y. Xu, P. Wang, W. Wang, and X. Zhu, "On optimizing traffic scheduling for multi-replica containerized microservices," in *52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, USA: ACM, 2023, p. 358–368.

[17] M. Straesser, P. Haas, S. Frank, A. Hakamian, A. van Hoorn, and S. Kounev, "Kubernetes-in-the-loop: Enriching microservice simulation through authentic container orchestration," in *Performance Evaluation Methodologies and Tools*, 2024, pp. 82–98.

[18] M. Chen, M. T. Islam, M. R. Read, and R. Buyya, "TraDE: Network and traffic-aware adaptive scheduling for microservices under dynamics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 37, no. 1, pp. 76–89, 2026.

[19] S. Frank, L. Wagner, A. Hakamian, M. Straesser, and A. van Hoorn, "Misim: A simulator for resilience assessment of microservice-based architectures," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 1014–1025.

[20] J. Wu, M. Xu, Y. He, K. Ye, and C. Xu, "Cloudnativesim: A toolkit for modeling and simulation of cloud-native applications," *Software: Practice and Experience*, vol. 55, no. 7, pp. 1185–1208, 2025.

[21] L. Leonini, E. Rivière, and P. Felber, "Splay: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09, USA, 2009, p. 185–198.

[22] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, 2021.

[23] M. Xu, Q. Zhou, H. Wu, W. Lin, K. Ye, and C. Xu, "Pdma: Probabilistic service migration approach for delay-aware and mobility-aware mobile edge computing," *Software: Practice and Experience*, vol. 52, no. 2, pp. 394–414, 2022.

[24] K. Fu, W. Zhang, Q. Chen, D. Zeng, X. Peng, W. Zheng, and M. Guo, "Qos-aware and resource efficient microservice deployment in cloud-edge continuum," in *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 932–941.

[25] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2021.

[26] Istio Authors, "Istio: An open source service mesh," https://istio.io/, 2025, accessed: 2025-11-02.

[27] Y. Gan and et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, USA: ACM, 2019, p. 3–18.

[28] A. Detti, L. Funari, and L. Petrucci, "μbench: An open-source factory of benchmark microservice applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 968–980, 2023.

[29] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, p. 23–50, Jan. 2011.

[30] Jaeger Authors, "Jaeger: an open-source distributed tracing platform for monitoring and troubleshooting complex microservices-based applications," https://www.jaegertracing.io/, 2025, accessed: 2025-11-02.

[31] Prometheus Authors, "Prometheus: An open-source technology designed to provide monitoring and alerting functionality," https://prometheus.io/, 2025, accessed: 2025-11-02.

[32] Fortio Authors, "Fortio: Istio's load testing tool," https://github.com/fortio/fortio/, 2026, accessed: 2026-06-09.

[33] B. Hubert *et al.*, "Linux advanced routing & traffic control howto," *Netherlabs BV*, vol. 1, pp. 99–107, 2002.

[34] Google Cloud Platform. (2026) Microservices demo: Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo. Accessed: June 13, 2026.

[35] R. N. Staff, "Ripe atlas: A global internet measurement network," *Internet Protocol Journal*, vol. 18, no. 3, pp. 2–26, 2015.