OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures

Abstract

Resource management for geo-distributed infrastructures is challenging for two key reasons: edge resources are scarce and heterogeneous, and situation awareness applications are highly dynamic due to client mobility and workload surges. State of the art schedulers that work well in a datacenter setting have limitations, owing to their centralized nature, both from the point of performance and features to match the requirements of such applications. We present OneEdge, a *hybrid* control plane that enables autonomous decision-making at the edge sites for localized, rapid application deployment. The edge sites collaborate with a logically centralized controller to facilitate coordinated multi-site scheduling and dynamic reconfiguration to deal with mobility, churn, and load spikes at the edge sites.

OneEdge includes features for respecting end-to-end (E2E) *service level objectives (SLOs)* of the applications and spatial awareness of the clients' locations in its scheduling decisions. A hierarchical monitoring component continually gathers statistics to drive reconfigurations when an application's SLOs are likely to be violated. The first contribution of OneEdge is its rich feature set that matches the requirements of situation awareness applications. The second contribution is the novel distributed state management that allows autonomous decision-making at the edge sites for localized resource allocations, in parallel with decision-making at the central controller for multi-site deployments. We evaluate OneEdge with a mix of applications using a multi-region Azure deployment. We show that OneEdge is able to deliver on the spatial affinity SLO for the applications. We also show that, compared to a centralized control plane, OneEdge reduces deployment latency by 66% for *standalone* applications, without compromising on E2E latency SLO violations compared to centralized.

ACM Reference Format:

. 2021. OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/XX.YYYY

1 Introduction

Situation awareness applications such as autonomous vehicles [11], drone navigation [12], AR-assisted driving [17], large-scale video analytics [1], and camera networks for safety and surveillance [33] continuously sense the environment and respond in real-time. In addition to being inherently geo-distributed, such applications are also both bandwidth intensive (*e.g.,* camera streams) and latency sensitive (*e.g.,* tight bound between sensing and actuation). Edge computing is a promising approach for meeting the resource needs of such applications, offering geo-distributed deployment of computational resources close to the sensor sources.

On the infrastructure front, the edge may encompass a range of heterogeneous computational resources, and we expect that the primary drivers will be micro-datacenters (μDC) with server-grade machines, a few racks per site, maintained by telecommunication providers and ISPs [23]. The combination of Cloud datacenters and such μDCs forms a computational continuum, as shown in Fig. 1.

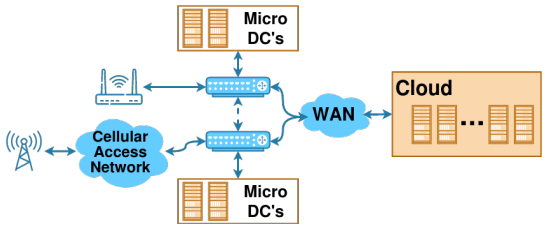


Figure 1: Edge-Cloud continuum: Micro-datacenters (μDCs) may be interconnected via dedicated fiber-optic links [27].

On the applications front, we identify two distinct application classes: *coordinated* and *standalone*. *Coordinated* applications feature multiple clients that share application state, hence different application instances require colocation and coordination. Examples of coordinated applications include collaborative assisted driving and geo-distributed multiplayer games (*e.g.,* Pokemon Go [31]). In contrast, *standalone* applications are limited to single-user instances . Examples include virtual reality and single-drone control.

Beyond their bandwidth and latency constraints, situation awareness applications pose unique requirements that distinguish them from Cloud-native applications. These requirements are: *R1: Autonomous control*—for single-site deployment of latency-sensitive *standalone* applications (§3.1.2), *R2: Coordinated control*—for multi-site deployment of *coordinated* applications (§3.1.1), *R3: Spatial affinity*—to support applications' location sensitivity, *R4: E2E latency SLO*—to support the necessary end-to-end latency SLO guarantees needed, and *R5: Dynamic resource reallocation*—needed for re-deployment of an application due to mobility and/or failures/resource scarcity at an edge site.

State of the art control planes like Kubernetes and its variants (*e.g.,* KubeEdge) do not meet these requirements. The primary reason is that such control planes were designed for throughput-oriented applications running in the Cloud. For

such applications, except for dynamic resource reallocation support (R5), the other requirements are either not applicable or easily met in a datacenter environment. Adapting the state-of-the-art to meet these requirements is non-trivial. For example, Kubernetes uses tag-based matching to select nodes for launching applications and does not support fine-grained latency-sensitive or location-sensitive application placement. Such requirements would have to be enshrined in application-specific controllers (Fig. 2a) running atop Kubernetes, pushing the burden on to the application developer to implement the required functionality independently.

OneEdge is an agile control plane designed to meet these requirements. Specifically, it allows edge sites to make autonomous scheduling decisions without central coordination for standalone applications. At the same time, to cater to the needs of coordinated applications which rely on global knowledge of application instances, OneEdge has a centralized component. For rapid autonomous control plane decisions without central coordination, the authoritative state is kept locally at each site. The central controller maintains an *eventually consistent* [29] aggregate state to make deployment decisions for multi-site coordinated applications. Such decisions are optimistic owing to the eventually consistent nature of the aggregate state and has to be ratified by the affected edge sites. OneEdge employs an enhanced two-phase commit protocol to ratify the deployment decisions with the affected edge sites. OneEdge exposes the right interfaces to the app developer to facilitate latency and location sensitive scheduling of the application components. The monitoring component of OneEdge ensures that the E2E latency SLO of an application is met, triggering migration of a client (*e.g.,* a connected vehicle) to a spatially appropriate application instance commensurate with the client mobility.

We make the following contributions:

- A novel hybrid control plane architecture for geo-distributed infrastructures that combines autonomous decision-making at edge sites to minimize deployment latency for *standalone* applications, with centralized decision-making for scheduling *coordinated* applications.
- Efficient optimistic concurrency control with an enhanced two-phase commit protocol that reconciles the central controller's eventually consistent state with the authoritative state that is distributed across edge sites.
- Intuitive interfaces for application developers to specify spatio-temporal constraints for applications, which are integrated in the control plane's scheduling decisions.
- The design and implementation of OneEdge, and an evaluation showcasing how its objectives are met.

Paper outline: §2 discusses the shortcomings of state-of-the-art control plane designs for situation awareness applications. §3 introduces a model for situation awareness applications and two concrete use cases. §4 presents OneEdge's key design principles and §5 its architecture. §6 evaluates OneEdge, using microbenchmarks and a mock-up of situation-awareness applications. Finally, we present concluding remarks and avenues for future work in §7.

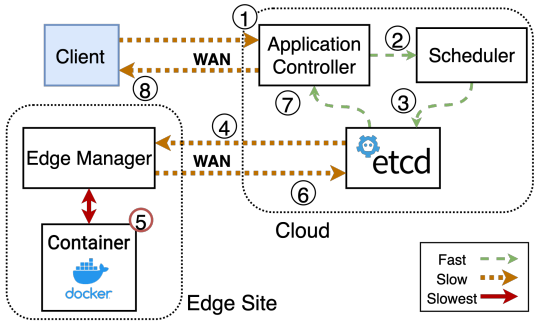
2 Limitations of Existing Control Planes

Cloud Resource Management. There is a large body of work in schedulers designed specifically for Cloud datacenters, ranging from monolithic [3, 28], partitioned [2, 5, 13, 30] to shared-state [6, 7, 15, 26] architectures. These systems are designed for the specific characteristics of datacenter environments: largely homogeneous computational resources with strong network connectivity between the control plane and the managed resources. Furthermore, they rely on a *shared authoritative state* (potentially replicated for redundancy and fault tolerance) to coordinate resource scheduling, which is updated by one or more distributed schedulers [26]. A shared state management approach is not scalable for edge-centric schedulers, as they need to reach the Cloud-resident shared state over a high-latency and unreliable network for every control decision. Disconnections and network partitions make it difficult, if not impossible, to achieve the desired attributes of autonomous decision-making and migrations.

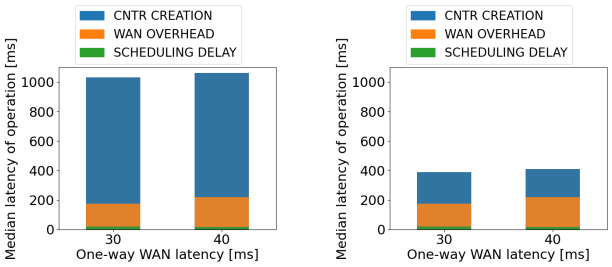
Geo-Distributed Resource Management. Alternative control planes for edge infrastructure such as KubeEdge [32] and Federated Kubernetes [18], inherit the centralized architecture of Kubernetes wherein all control plane decisions are driven by a central controller.

To underscore the deployment overhead of the current state-of-the-art for situation-awareness applications, we conduct the following experiment with Kubernetes. Fig. 2a shows the experimental set up: a client of a situation-awareness application, a desired edge site for launching the application for the client, and Kubernetes scheduler in the Cloud. The color-coded arrows show the relative latencies for each of the control flow actions. The control flow for launching the application at the edge site depicted in Fig. 2a is as follows. ① and ②: The application's client communicates a site-specific deployment request to a dedicated application controller in the Cloud hosting the Kubernetes scheduler. ③: Kubernetes makes the scheduling decision and enters it into the *etcd* database. ④ to ⑥: The edge site picks up the scheduling request, launches the needed application containers, and informs Kubernetes. ⑦ and ⑧: The application controller is apprised that the application is ready to be launched and notifies the client, which can now start interacting with the deployed application on the edge site.

To showcase the best case deployment latency (*i.e.,* no queuing effects due to other requests pending at the scheduler) for Kubernetes for different controlled settings of WAN latency, we emulate all the entities (Client, Cloud, and Edge Site) involved in the control flow shown in Fig. 2a as individual VMs inside an instance of an Azure region [20]. Every



(a) Workflow for app deployment on an edge site using Kubernetes.



(b) Latency breakdown with container cold start.

(c) Latency breakdown with pre-warmed containers.

Figure 2: Experimental results with Kubernetes.

WAN hop shown in Fig. 2a incurs a set latency controlled through the Linux *tc* [19] utility. Fig. 2b and Fig. 2c show the mean end-to-end deployment latency for different settings of WAN latency. Fig. 2b is for deploying containers from scratch (cold start), while Fig. 2c is for pre-warmed containers. The bar graphs show the breakdown of the latency into individual components. As can be seen from Fig. 2b, the container startup time dominates the end-to-end application deployment latency. However, significant ongoing research efforts are focused on addressing the high cost of cold starts by keeping pools of pre-warmed containers [22] to avoid this overhead. Fig. 2c demonstrates that once the cold start effect is mitigated (using pre-warmed containers), the overhead of WAN traversal becomes the primary deployment latency determinant. For example, with pre-warmed containers and a 40ms one-way WAN latency, the WAN overhead accounts for 49% of the deployment latency. We observe a similar trend with KubeEdge (but with higher latency due to additional book-keeping overheads), since it has a similar deployment workflow.

Ensuring low-latency control plane actions is important for situation awareness applications both to get the application started initially, as well as for reconfiguration decisions in response to client mobility or resource scarcity. If not executed quickly, control plane actions can result in E2E SLO violations for the applications. Besides the latency concern of placing multiple WAN traversals on the critical path of application deployments, state-of-the-art schedulers also do not natively cater to the requirements of situation awareness applications (§1), specifically meeting their E2E latency

Type of Scheduler	Requirement				
	R1	R2	R3	R4	R5
Cluster Monolithic (*e.g.*, Kubernetes)	N*	N*	N*	N	Y
Cluster Partitioned (*e.g.*, Mesos)	N*	N	N*	N	Y
Cluster Shared-state (*e.g.*, Omega)	N*	N	N*	N	Y
Geo-dist. Centralized (*e.g.*, KubeEdge)	N*	N	N	N	Y
Geo-dist. Decentralized (*e.g.*, Foglets)	Y	N*	Y	Y	Y
Geo-distributed Hybrid (OneEdge)	**Y**	**Y**	**Y**	**Y**	**Y**

Table 1: Comparison of schedulers with respect to the requirements of situation awareness applications. The requirements are **R1: Autonomous control**, **R2: Coordinated control**, **R3: Spatial affinity**, **R4: E2E latency SLO**, **R5: Dynamic resource reallocation**. N = requirement not met; N* = requirement incompatible with system's architecture.

SLOs, and respecting spatial affinity considerations. Table 1 summarizes the ability (or lack thereof) of state-of-the-art control planes for meeting the requirements of situation awareness applications.

The application controller in the Cloud (Fig. 2a) acts as a layer above Kubernetes to instruct the scheduler on its desired placement decisions. It is a burden on the developer to build an application controller for each situation awareness application. This begs the question, *could we build a generic abstraction layer on top of Kubernetes that caters to the requirements (Table 1) as a viable solution?* Unfortunately, as noted above, existing mechanisms in state-of-the-art schedulers such as Kubernetes do not inherently cater to such requirements. Therefore, we take a clean-slate approach to systematically address all these requirements with OneEdge.

3 Situation Awareness Applications

Situation awareness applications convert geo-distributed 24×7 sensed data to actionable knowledge at *computational perception* speeds and are highly latency-sensitive. This section formalizes a general application model for situation awareness applications and highlights two concrete use cases.

3.1 Application Model

Situation awareness applications can be modeled as a pipeline of components (*e.g.*, detection, filtering, and sub-regional view, as shown in Fig. 3). The client (*e.g.*, a vehicle) feeds input data into a pipeline. Each component processes data generated by the upstream component and generates output data to be consumed by the downstream one. Additionally, actionable information generated by a component can be communicated back to the client directly.

Representing the application as a pipeline gives the control plane the flexibility to split application components across multiple servers within a μDC or even across μDCs, so long as the application's SLOs are not violated. A pipeline model allows sharing key stages across clients to enable cross-client coordination (*e.g.*, sub-regional and regional views in Fig. 3). We describe a representative example of a *coordinated* and a *standalone* application.

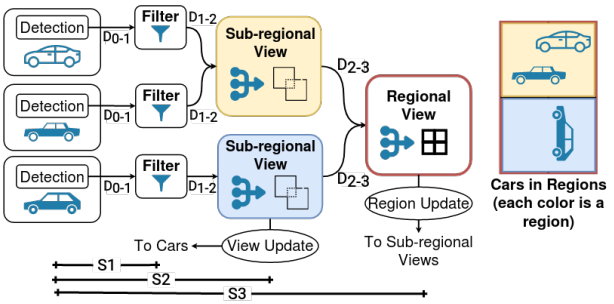


Figure 3: An exemplar of situation awareness application – Connected Vehicles. Cars in the same spatial locale have their individual views fused by the sub-region view; Region view fuses sub-region views of adjacent spatial locales.

3.1.1 Coordinated Application: Connected Vehicles

We consider a connected vehicle scenario (Fig. 3) modeled as a pipeline. Each vehicle (*i.e.,* client) uses a lidar sensor and on-board processing to generate a list of objects it can *see* in its immediate field of view. The individual views from multiple vehicles in close spatial proximity of one another are aggregated to create a composite view (*sub-regional view*), which helps reveal objects missed by the individual views due to occlusions. The fused composite view is made available to the vehicles in the same spatial proximity so that each vehicle can take better decisions for lane control and collision avoidance. Different disjoint subsets of vehicles in different spatial locales have their views fused together as independent sub-regional views. The sub-regional views may be aggregated at the next pipeline stage to create a regional view to further improve vehicular safety and traffic analyses. The mobility of the vehicles necessitates dynamic, constantly evolving associations of vehicles with spatial sub-regions.

3.1.2 Standalone Application: Drone Navigation

Control of a single autonomous drone is an example of a standalone application. The application model is much simpler, as it does not share any application component or state across clients. As there are no sharing requirements, the best placement for standalone apps, from a network latency and bandwidth perspective, is on the geographically closest edge site, both for placement and mobility-triggered migration.

3.2 Application-level SLOs

For situation-awareness applications, SLOs are best defined in terms of tolerable latencies. For example, with reference to Fig. 3, the "Sub-Regional View" stage has a more stringent latency requirement compared to the "Regional View" stage. The tolerable latency for a particular stage is the composite of all the latencies leading up to that stage, including network latency, queuing, and computation times. The cost of network communication also depends on the data exchanged between the stages: *e.g.,* the "Filter" stage receives raw data from on-board sensors and needs a high-bandwidth connection to the raw data streams; while the "Regional View" stage, which aggregates metadata from sub-regions, has a much smaller bandwidth requirement.

Since an application is expressed as a pipeline of components, it is convenient for its developer to specify the SLOs in terms of two parameters per pipeline stage: (a) acceptable *staleness*, and (b) *rate of data* production. Both parameters depend on the application's semantics.

Fig. 3 shows the tolerable staleness for stage i using the notation S_i. S_i denotes the worst-case acceptable composite latency at the input of stage i. D_{i-j} refers to the production rate of data items communicated between stages—*e.g.,* the data rate between the "Filter" and "Sub-Regional View" stages D_{1-2} is the objects detected per unit time (bytes/sec).

Presenting the SLOs in terms of data staleness and data rates gives flexibility to the control plane in placement decisions, taking into account the μDCs' resource capacities to accommodate each stage's CPU and memory needs, and the bandwidth required for communication between the stages. Concretely, the E2E latency SLO of an application is equal to $\sum_{i=1}^{n} S_i$, where n is the number of stages in the pipeline.

Another quality of interest for *coordinated* applications is *spatial affinity*, which we define as *the application's intent to share state among a subset of clients based on geographical proximity, i.e., Area of Interest (AoI).* Each application may have its logic for defining an AoI for its clients. For example, in the connected vehicles application (Fig. 3), an AoI may be the area covering a busy intersection. Spatial affinity is an application SLO that the scheduler will use in its decisions.

To quantify how well an application deployment chosen by a scheduler matches the application's spatial affinity SLO, we propose a new metric, namely, *spatial alignment.* With respect to Fig. 3, if there are n vehicles in a given AoI (*i.e.,* the AoI's *current* spatial affinity is n), the metric should quantify the percentage of vehicles (*i.e.,* a fraction of n) whose individual views are fused by the sub-regional view component. Thus, we define *spatial alignment* for an AoI as follows:

$$spatial\ alignment = \frac{\#\ of\ clients\ in\ AoI\ sharing\ the\ app\ pipeline}{\#\ of\ clients\ in\ AoI}.$$

A perfect scheduler maps all clients with the same spatial affinity to the same pipeline, *i.e.,* spatial alignment = 1.

The scheduler's operation uses both metrics: E2E latency SLO and spatial affinity SLO, and aims to maximize spatial alignment and minimize E2E latency SLO violations.

4 Challenges and Key Design Principles

In this section, we elaborate on the challenges associated with the five key requirements for control planes managing situation awareness applications on geo-distributed infrastructures. We then introduce key control plane design principles that allow OneEdge to overcome these challenges.

Challenge 1: Situation awareness applications have inherently strong topological semantics. First, strict latency SLOs

are tightly associated with an application's deployment location relative to each client's physical location, as network traversals account for a significant fraction of each serviced request's E2E latency. Second, spatially proximal clients of the same coordinated application should be digitally colocated to enable essential application state sharing. While topological information should be integral to scheduling decisions, that is not the case for existing control planes.

Challenge 2: Standalone and coordinated applications impose conflicting requirements on the control plane. Autonomous deployment necessitates distributed state and independent decision-making based on locally available state, while the need for balanced load on widely fragmented resources and for inter-site client coordination necessitates an orchestrating entity with global state visibility.

Challenge 3: As application pipelines (Fig. 3) may span multiple edge sites, site-specific performance monitoring alone is insufficient to provide E2E latency guarantees.

To overcome these challenges, OneEdge's architecture builds on three main design principles:

Principle 1: Client geolocation and application latency requirements are exposed as *first-class citizens* to the control plane. This information is contained in each client's application deployment request (*i.e.,* client's GPS location), and the control plane's state bookkeeping is geospatially organized.

Principle 2: A two-level hybrid structure to reconcile the conflicting need for both distributed and centralized state. The hybrid control plane architecture comprises an autonomous controller per μDC and an overarching centralized controller playing complementary roles to meet the needs of both standalone and coordinated applications. Autonomous per-site controllers maintain the site's authoritative state and allow instant deployment of standalone applications without interacting with the centralized controller. The centralized controller maintains an *eventually consistent* view of the global state, which is leveraged for cross-site application pipeline deployment and for off-the-critical-path resource reallocation decisions for load balancing purposes.

Principle 3: Application deployment decisions should be primarily SLO-driven rather than just resource-driven. Preserving application E2E latency guarantees should be a responsibility of the control plane. Thus, the control plane should not only make placements based on performance targets, but also enforce continuous compliance via hierarchical monitoring. Per-application latency metrics should not only be locally collected at each per-site controller, but also periodically aggregated at the centralized controller, to assess E2E SLO compliance. Upon SLO violation detection, the control plane reassesses resource provisioning decisions and/or migrates applications depending on the source of SLO violation (*i.e.,* due to resource scarcity or client mobility).

5 OneEdge System Architecture

We now present the architecture of OneEdge, a hybrid control plane that embodies the requirements outlined in Sections 1 and 2 and builds on the design principles in §4.

5.1 Overview

Fig. 4 depicts OneEdge's high-level architecture, comprising two top-level entities: *site* and *controller*. A site is a self-managed instance of a geo-distributed infrastructure (*e.g.,* μDC, Cloud datacenter) that contains computational resources and cooperates with the controller in control-plane decision making. The controller is a logically centralized entity that makes system-wide deployment decisions for application pipelines spanning more than a single edge site, or to adjust deployment decisions that were autonomously made at individual sites, for load balancing reasons.

Client requests to launch an application on the edge infrastructure are always directed to the client's geographically proximal site by using a standard discovery service [21]. The application is tagged as "standalone" or "coordinated" by the developer. The site agent determines whether to handle it locally or forward it to the central controller. Standalone applications are handled locally, avoiding WAN traversals and centralized coordination, unless resource constraints prevent local deployment. Deployment requests for coordinated applications are always forwarded to the central controller.

The *central controller* monitors and orchestrates the aggregated infrastructure's state; it plays both a *proactive* and a *reactive* role, on and off an application deployment's critical path, respectively. For coordinated applications, it determines the matching of each client to application instances, and the placement of application pipeline components to specific sites. In the background, it also monitors each site's resource usage and each application's delivered performance compared to its SLO, to drive resource re-allocation and application migration decisions.

5.2 Edge Site Components and Operation

Each site comprises four components (Fig. 4, right): the site agent, the monitoring subsystem, the container agent, and the container runtime (which relies on Docker [8]). The site agent is the site's local resource manager and coordinator with the central controller. It receives deployment requests from the clients within the site's region and from the central controller. Each deployed application component is associated with a container agent. Each application's container agent handles any inter-site communication between application pipeline stages spanning multiple sites. Finally, the monitoring subsystem continually gathers metrics about the site's resource usage and metrics pertaining to the SLOs of the application components hosted on that site. OneEdge's monitoring subsystem is further discussed in §5.3.2.

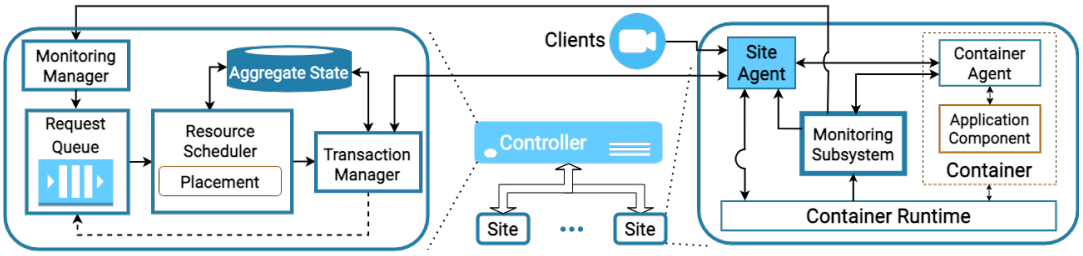


Figure 4: OneEdge's System Architecture. A central controller (left blow-up) coordinates with all the edge sites (right blow-up).

Deflection. In general, a site forwards new deployment requests for coordinated applications to the central controller and autonomously handles the local deployment of standalone application requests. When a site's resources are highly utilized, even standalone application requests cannot be locally served and have to be *deflected* to the central controller. The central controller will then deploy the application on a nearby edge site that can meet the application's SLOs. A site's deflection policy is controlled by two parameters: *threshold* and *percentage*. When the resource commitment at a site exceeds a specified *threshold*, up to the indicated *percentage* of new client requests will be deflected to the centralized controller for alternative placement. A fully saturated edge site deflects all new requests.

5.3 Controller Components and Operation

OneEdge's central controller (Fig. 4, left) plays two crucial roles. First, it determines the placement of cross-site application pipelines, a need typically associated with *coordinated* applications. Second, it continuously monitors the entire infrastructure for significant load imbalances and E2E application SLO violations. When such incidents are detected, the controller will make reallocation and/or migration decisions to ameliorate the problem. The two roles are fulfilled by the controller's proactive and reactive policies, respectively.

5.3.1 Proactive Policies

Fig. 4 (left) shows the central controller's workflow for handling deployment requests received from site agents (mainly for *coordinated*, but also for *standalone* applications due to deflection). Additionally, the monitoring subsystem may also insert reconfiguration requests into the request queue to avoid potential SLO violations detected through the monitoring statistics (see §5.3.2). The *resource scheduler* processes the requests in order, making placement decision to match each request's requirements: (i) type—standalone or coordinated; (ii) E2E latency SLO—inferred from the application pipeline to be launched similar to Fig. 3; and (iii) spatial affinity—inferred from the request-initiating client's GPS location. We use the metrics defined in §3.2 to quantify E2E latency and spatial alignment in the proactive scheduling decisions.

To facilitate resource sharing while respecting application SLOs, OneEdge uses offline profiling to generate a *resource requirement profile (RRP)*, for each application component that may be shared across client requests. The RRP specifies the resource commitment needed for each stage of an application pipeline as a function of the number of concurrent clients. RRP is used by both the site agent and the central controller's resource scheduler to make allocation decisions and is therefore populated in both sets of components.

The *placement* module in the resource scheduler selects the site(s) to launch the application pipeline for the client request. In making this placement decision, resource scheduler consults the *aggregate state* which is the composite of the resource commitments of all the μDCs. Due to the autonomous decision making at the μDCs, the *aggregate state* may not be up to date. Keeping the *aggregate state* eventually consistent is a principle (§4) we adopted to give local autonomy for making rapid deployment for latency-critical standalone applications. Placement decisions are made optimistically, under the assumption that *aggregate state* is up to date. Our approach follows the "think globally, act locally" conventional wisdom in distributed systems.

The role of the *transaction manager* in the central controller workflow is to launch the placement decision as an atomic 2-phase commit (2PC) transaction, because it may affect multiple sites in the case of coordinated applications. At the end of the first phase, the transaction manager will know if the placement decision has been accepted by all the affected sites. In this case, the second phase of the transaction is to confirm the placement decision to the affected μDCs. If any of the affected sites reject the decision in the first phase, the transaction manager sends an abort message in the second phase to all the affected sites, and updates the *aggregate state* using the authoritative state information received from the sites. The μDCs update their internal authoritative states upon receiving the abort message. After an abort, the request is re-enqueued in the request queue with a higher priority.

Since the resource scheduler updates the *aggregate state* after completing every request and uses the new *aggregate state* to process the next request, an *invariant* that should be maintained by the transaction manager is that the transactions should *appear* to be applied on the sites *serially*. We discuss optimizations to preserve this requirement without compromising performance in §5.3.3.

5.3.2 Reactive Policies

While the central controller's proactive policies place applications so that E2E SLOs are met and resource utilization across the infrastructure is balanced, continuous adaptations may be needed for various reasons. First, autonomous deployments of *standalone* applications at edge sites can cause utilization imbalance and increased resource pressure at individual edge sites. Second, client mobility can cause frequent load shifts between edge sites serving the same application. Such events can lead to application SLO violations, thus the controller needs to reevaluate its resource allocation and application placement decisions continuously. At the heart of these reevaluations lies OneEdge's monitoring subsystem.

Hierarchical SLO Monitoring. Each site has its own monitoring subsystem, which periodically aggregates each locally running application's metrics of interest (*e.g.,* per-stage execution time). For *standalone* applications, the aggregated statistics are conveyed to the site agent. If the site agent detects application SLO violations, it attempts to alleviate the issue locally by allocating additional resources to the suffering application's containers. If that is not possible (*e.g.,* no local resource availability or client mobility), the agent notifies the central controller, taking actions at a global scale.

For *coordinated* applications spanning multiple edge sites, per-site statistics have to be combined to determine potential SLO violations. The site agent forwards the locally aggregated statistics (*e.g.,* execution times, data production rates, queuing between stages) to a preselected *leader* site, which hosts some of the application pipeline's stages. The leader site summarizes the collected statistics and sends a digest to the central controller's *monitoring manager*. The *monitoring manager* uses this information to determine if an application reconfiguration (*e.g.,* increase resource allocation at each involved site or migrate application stages) is necessary; if so, it generates a new request in the *request queue*.

Dynamic Resource Reallocation. The first reaction to detected SLO violations is an incremental resource allocation increase for the container(s) hosting the target application pipeline's stages. The controller uses the RRP (§5.3.1) as an application-specific guide to define the extra resources that need to be allocated to avoid the SLO violation, as follows. The allocation for the affected application is increased by λ *dummy* clients, where λ is a configuration parameter (a small positive integer). Suppose C is the number of clients currently served by the application pipeline. In that case, the allocation is increased to that needed for $C + \lambda$, using RRP to identify the required resources corresponding to the new number of clients. Incremental allocation provides the agility necessary to react quickly to surges in resource needs.

The second reaction knob is the migration of the application's pipeline stages from the site experiencing the load spike that caused the SLO violation to other sites. The central controller uses three inputs to guide this action: (a) knowledge of spatially proximal sites to the affected site, (b) the resource commitments at these proximal sites (available from *aggregate state*) to ensure load balance, and (c) the requirements of the application components to be migrated.

5.3.3 Performance Optimizations

Enhanced 2PC. The central controller uses a 2PC protocol to deploy *coordinated* applications spanning multiple sites. The traditional semantics of 2PC would abort a transaction T_i if there is a mismatch between the controller's aggregate—but often stale—state when T_i was generated by the resource scheduler and the site's authoritative state when T_i is processed at the site. To avoid unnecessary aborts, OneEdge leverages the observation that a transaction need not abort as long as the sum of transaction's requested resources and currently reserved resources does not exceed the site's resource capacity (Omega [26] exploits a similar idea in a datacenter setting). When such conditions are met, instead of aborting the transaction, the site agent updates the authoritative state with the transaction's allocation request during phase one of the protocol, and informs the central controller of the actual site resource commitments to update the *aggregate state*.

A second optimization to conventional 2PC reduces the latency on the critical path from two WAN traversals to one. In the first phase, a site replying affirmatively to a deployment request also optimistically allocates the requested resources. If the controller receives affirmative responses from all affected sites, it notifies the client in parallel with the execution of the second phase. Thus, the WAN latency for the second phase can be overlapped, as the site can start receiving actual data plane actions from the client ahead of the second phase's completion. If the transaction is aborted, the second phase frees each site's optimistically committed resources.

Transaction Pipelining. It is reasonable to expect geospatial diversity across successive client requests arriving at the central controller. Successive transactions affecting *disjoint* sets of sites are independent; so, the transaction manager could execute them in parallel. However, for the correct operation of the resource scheduler, transactions should appear to be executed serially by the transaction manager. One way of exploiting parallelism and preserving this ordering invariant is to enforce ordering at the destination sites.

The following conditions should be met at the destination site to ensure that transaction order is preserved while executing transactions (which may or may not be independent of each other) in parallel: (1) successive transactions that affect the same site should be processed by the site *in the scheduler's order of generation*, and (2) a transaction abort should correctly restore the authoritative state at the site before that site processes subsequent transactions.

We denote $D(T_i)$ to represent the dependency set of transactions T_j for which $T_j \rightarrow T_i$. Similarly we denote $AD(T_j)$ to

denote anti-dependency, *i.e.,* the set of transactions T_i such that $T_j \rightarrow T_i$. A transaction T_i is eligible to be processed at each affected site so long as all the transactions in its dependency set $D(T_i)$ have been completed. Every transaction T_i sent to a site contains $D(T_i)$ and $AD(T_i)$.

The site will not process T_i unless it has already received and processed T_j. The completion of T_j will trigger the deletion of all the incoming edges from the transactions in $AD(T_j)$, possibly allowing some of them to become eligible for processing. If T_j is aborted, the *aggregate state* is rolled-back accordingly. Further, this abort will trigger a cascading roll-back of the pending transactions that transitively depend on T_j (*i.e.,* starting from the members of $AD(T_j)$). These aborted transactions will result in a re-submission of the associated control plane requests to the request queue. However, aborts are uncommon, because, per our *Enhanced 2PC* protocol, they only occur if the site's resources have been depleted (and not due to a mere mismatch between the central controller's and the site's state of resource availability).

Additionally, to prevent queue build up at the destination sites, we use a *windowing* technique, which limits the maximum number of outstanding transactions that can be sent to a given site. The limit ensures that a site is not overloaded.

5.3.4 Fault Tolerance and Scaling up

Fault tolerance for the centralized controller is provided in a standard manner. We assume that the central controller runs in a robust environment (*e.g.,* Cloud datacenter). While the server that hosts the central controller may fail, it is improbable for the entire datacenter to fail. Therefore, the approach to fault tolerance is to have a secondary instance of the central controller running in tandem with the primary in another server. All the pertinent information involved in the primary workflow (§5.3.1) is replicated, including in-flight transactions. On a primary failure, the secondary takes over and rolls back to an *aggregate state* comprising only complete transactions, by issuing aborts for all the in-flight transactions and resubmitting them as new requests.

Given our scope, we choose a simple design for the central scheduler. In a full-scale scenario, the central scheduler can still become a bottleneck, calling for a more sophisticated design that allows scaling its performance. Well-known Cloud techniques [26] can be directly applied to address such challenges; this is part of our future work (§7).

6 Performance Evaluation

OneEdge is implemented in C++11 on Ubuntu 18.04. Each application component is dynamically linked in to a base OneEdge container image built using the Docker framework. We use MongoDB to store the *transaction manager*'s *aggregate* state, and ZMQ for communication among the system's distributed components. The evaluations we report in this section aim to verify the following hypotheses:

1. Transaction pipelining and enhanced 2PC (§5.3.3) are effective in improving OneEdge's performance (§6.2.2).
2. OneEdge's hybrid nature delivers lower-latency placement decisions compared to a centralized control plane for *standalone* requests (§6.2.3).
3. OneEdge strikes a good compromise between deployment latency for *standalone* requests and load balance across latency-equivalent edge sites (§6.2.4).
4. OneEdge is similar to a centralized control plane at meeting application SLOs, while achieving significantly lower deployment latency for *standalone* applications (§6.3).

We derive a centralized control plane baseline by configuring OneEdge's site agents to deflect *all* incoming requests at edge sites to the central controller. The resulting centralized baseline is functionally similar to KubeEdge [32], enhanced with the pipelined control plane actions described in §5.3.3 and satisfying E2E latency constraints of applications.

6.1 Experimental Platform

Azure Regions. For the microbenchmarks and end-to-end application evaluation, we emulate a geo-distributed edge infrastructure with sites in multiple metropolitan areas. To do so, we use resources in five Azure regions: WestUS, WestUS 2, CentralUS, South Central, and East US. We host the central controller on the East US region and use each of the remaining regions to emulate a different metropolitan area. In each region we create multiple VMs, and each VM represents a μDC (*i.e.,* an edge site) located within that metropolitan area. We used "Standard D16s_v4" VMs, each featuring 16 vcpus and 64 GiB memory. Emulated clients are tied to a specific metropolitan area and are hosted in the corresponding Azure region. Clients exhibit mobility only inside their associated metropolitan area, but not across metropolitan areas.

6.2 Microbenchmarks

In this section, we stress-test OneEdge by executing all the control plane actions without any application running or any actual resource allocations. To stress-test at scale, we emulate μDCs and parameterize the container runtime within each μDC. Further, we synthesize the client workload presented to OneEdge to drive the controlled experiments.

6.2.1 Experimental Setup

Control-plane Parameters. OneEdge consists of one *Central Controller,* multiple *μDCs*, and multiple *Clients* (Fig. 4). We use a simplified placement logic to stress test our performance optimizations. The controller's resource scheduler uses two common heuristics: round-robin placement across μDCs (in the same metropolis) to improve allocation balance, and collocation of an application's pipeline stages on the same μDC, if capacity allows, to improve application E2E latency. We set resource scheduling latency to 2 ms, matching the best-case performance of our resource scheduler implementation used in the end-to-end study (§6.3).

Parameter	Value
Container Startup/Update	583 ms/25 ms
One-way WAN latency	20 ms
Window Size	100
Resource Scheduling Latency	2 ms
Modeled per-μDC resource capacity	4096 cores, 8 TB memory
Per-request resource allocation	1 core, 512 MB memory
C-App Exponential β range	10–100 s
C-App Poisson λ range (per μDC)	1–8 s^{-1}
S-App Exponential β range	100–300 s
S-App Poisson λ range (per μDC)	2–25 s^{-1}

Table 2: Summary of parameters for microbenchmarks.

Emulated μDCs. In the emulation, although each μDC is represented by a single VM, its resource capacity is modeled as comprising 32 servers with 128 cores and 256GB of DRAM per server. Note that the modeled per-μDC capacity is only for book-keeping purposes during the microbenchmark experiments—resources are not actually allocated.

Container Runtime. To parameterize the times associated with the management of an application pipeline on a μDC, we measure the Docker container runtime used in the site agent's implementation (Fig. 4). The measured mean container deployment time (consisting of a simple application and its container agent library) is 583ms, with a std. deviation of 143ms. The measured mean time for updating an already deployed container's resource allocation (CPU-set and memory limit [9]), is 25ms, with a std. deviation of 4ms. We use these results to parameterize the μDC's reaction time upon every deployment request in the microbenchmarks.

Workload Characterization. We generate a synthetic workload with a mix of deployment requests for *coordinated* and *standalone* applications (abbreviated as *C-App* and *S-App*, respectively). We construct the synthetic workload by studying the cellular mobility of cars in San Francisco (SF), using the SF cabs dataset [24] and locations of cellular towers in SF [4]. We use k-means to group the cellular towers into 32 clusters and select each cluster's resulting centroid as a μDC's location. Clients send allocation requests to the geographically closest μDC, hence client-μDC communication does not incur WAN latency. As each mobile client's closest μDC changes over time, *C-App* clients trigger migration requests, while *S-App* clients send an allocation request to the new closest μDC. Since taxis represent a small subset of a city's whole car fleet, we overlay the 23 days worth of data in the SF cabs dataset to increase the number of simultaneously active cars. We model each μDC's request arrival using a Poisson distribution and each client's connection duration to a given μDC using an exponential distribution, and instrument these distributions with the ranges of client inter-arrival and connection duration extracted from our enlarged overlaid dataset, for both *C-App* and *S-App*. Table 2 summarizes the microbenchmarks' parameters.

We use the parameters extracted from SF to model one metropolitan area's workload. To conduct larger-scale experiments with multiple metropolitan areas, we replicate the edge infrastructure (using additional Azure Regions), and the above SF workload to every additional metropolitan area we model.

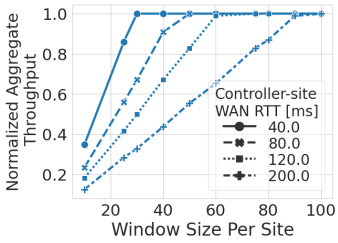


Figure 5: Impact of Pipelining on Aggregate Throughput.

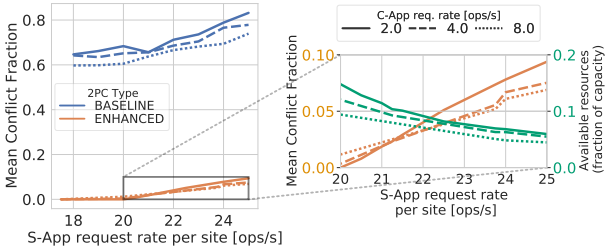


Figure 6: MCF of baseline and enhanced 2PC for constrained resources at a μDC and typical *S-App* and *C-App* request rates from the SF Cabs dataset (Table 2). The blow-up shows the increase in MCF and the μDC's remaining available resources for enhanced 2PC at higher request rates.

6.2.2 Evaluation of OneEdge's Optimizations

We evaluate OneEdge's optimizations (§5.3.3)—transaction pipelining and enhanced 2PC—using a single μDC.

Transaction Pipelining. The windowing mechanism in the *transaction manager* (§5.3.3) aims at mitigating the effect on the throughput of placement requests due to the WAN round-trip time (RTT) between the controller and the μDC. The choice of window size is a function of the WAN RTT. To study the effect of the window size on OneEdge's throughput, we construct an experiment consisting of only *C-App* requests, where the request generation rate is set to correspond to the maximum throughput achievable by the central controller without any queuing delays (*i.e.,* equivalent to having a WAN RTT of zero and no transaction aborts). Fig. 5 plots the effect of window size on OneEdge's aggregate throughput (normalized to the maximum throughput achievable) for different WAN RTT settings. Naturally, the minimum window size required to maximize throughput grows as a function of WAN RTT. For example, a WAN RTT of 40 ms requires a minimum window size of 50 transactions to reach the maximum throughput. This result highlights the need to batch multiple placement requests to mitigate the negative impact of WAN RTT from the controller to the μDC. Therefore, we have chosen a conservative window size of 100 for all the remaining microbenchmark experiments.

Enhanced 2PC Protocol. This optimization aims at avoiding unnecessary rollbacks of placement requests due to state

mismatch between the central controller and a μDC (§5.3.3) by performing state reconciliation. To evaluate the technique's effectiveness, we use the metric *mean conflict fraction (MCF)*, defined as *the average number of conflicts per successful transaction*. A score of zero represents no conflict, while a non-zero value indicates the number of aborts that happen for each successful transaction. For this evaluation we use a mix of *C-App* and *S-App* requests and disable *deflection* to deterministically control which requests are handled at the central controller as opposed to locally at the μDC.

In this evaluation, we focus on scenarios with high μDC resource commitment. We therefore use the higher arrival rate ranges from Table 2: we vary the *C-App* and *S-App* request arrival rates between 4–10 and 15–25 requests per second, respectively, while keeping the *C-App* and *S-App* client durations fixed at 50 and 200 seconds, respectively.

Fig. 6 shows the MCF for the enhanced 2PC compared against baseline 2PC over a range of request arrival rates. The MCF of baseline 2PC is consistently higher than the enhanced 2PC, which only becomes non-zero at high arrival rates, when the resource commitment at the μDC is sufficiently high to cause transaction failures due to capacity overcommitment. For example, at an *S-App* rate of 20 req/s, the μDC resource commitment is 88–90% of its total capacity and it is only from this point on that the MCF increases for enhanced 2PC. Fig. 6's inset plot is a blow-up of the enhanced 2PC results to show the increase in MCF with increasing request rates. Even at an observed capacity of 95% (corresponding to the largest *S-App* arrival rate shown in the graph), the MCF for enhanced 2PC is an order of magnitude lower compared to baseline 2PC. On realistic deployments with multiple μDCs, the central controller can further reduce the probability of capacity-caused conflicts by avoiding scheduling new requests on μDCs with resource commitments over a threshold (*e.g.,* 80%).

The higher the MCF, the higher the probability of failure, hurting the latency of *C-App* requests because their successful execution requires repeated scheduling attempts across the WAN. The MCF trends can be used to extrapolate the probability of a failure for applications deployed across n μDCs: the probability of failure is $1-(1-f)^n$, where f is the probability of transaction failure on a single μDC, which equals $MCF/(1+MCF)$. A higher MCF increases the likelihood of failure, which means that the enhanced 2PC's positive effect is multiplicative in the multi-μDC scenario.

The results from the windowing and enhanced 2PC experiments validate our first hypothesis regarding the effectiveness of OneEdge's optimizations in improving performance.

6.2.3 Control Plane Effect on Standalone Applications

Next, we quantify the performance advantage of OneEdge over a centralized control plane. As we noted earlier, the centralized baseline is similar to KubeEdge [32] in terms of control plane actions. The metric of interest is latency per

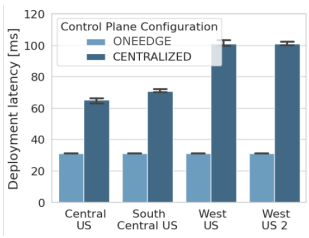


Figure 7: *S-App* deployment latency: OneEdge vs. Centralized control plane.

S-App deployment request. The experiment uses a μDC in each of the four metropolitan regions (§6.1). Deployment requests originate from the locales of all four μDCs, following the parameters in Table 2. For this experiment, we turn off deflection to fully control where each request is processed.

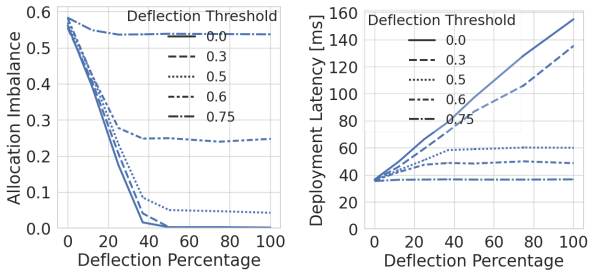
Fig. 7 shows the deployment latency of *S-App* requests from each of the four metropolitan areas considered in our scenario. We chose scenarios with low load to avoid queue buildup in the scheduling entity. We categorize the requests based on their origin μDC. Centralized incurs higher deployment latency than OneEdge, and is higher for μDCs further away from the central controller. In contrast, OneEdge incurs a constant low latency irrespective of the WAN latency between edge and controller, as it depends only on the container's allocation update latency (25 ms as per Table 2).

These latency results corroborate our second hypothesis regarding the advantage of OneEdge over a centralized control plane in terms of deployment latency for *S-App* requests.

6.2.4 Latency/Load-balance Trade-off

OneEdge's deflection mechanism (§5.2) enables the central controller to load-balance *S-App* applications across μDCs that are equivalent in terms of providing the latency requirements of the requesting client. We construct a microbenchmark to evaluate the trade-off between achieving low latency for *S-App* requests and the desired property of resource allocation balance across latency-equivalent μDCs. The metric used is *allocation imbalance*, defined as *the difference between the highest and lowest resource commitments among the latency-equivalent μDCs at a given time*. Hence, a value of zero for this metric indicates perfect load balance.

We evaluate a setting of 8 μDCs in the same metropolitan area, with all of them equally capable of meeting the latency requirements of all the emulated client requests emanating from that region. We calculate the allocation imbalance metric for this set of μDCs. When the central controller receives a deflected request, its placement algorithm selects the μDC with the lowest resource commitment towards a more evenly balanced load. The workload consists of only *S-App* requests which are skewed such that if all the requests are handled locally (*i.e.,* no deflection), 50% of the μDCs in each cell would have 80% of their resources committed, while the remaining



(a) Allocation imbalance. **(b)** Deployment latency.

Figure 8: Trade-off between allocation imbalance and deployment latency for *S-App* requests handling at proximal μDC vs. at central controller.

50% of the μDCs would have 20% of their resources committed on an average. Thus, this workload without deflection results in an allocation imbalance of 0.6.

We present results for the 8 μDCs emulated in the West US Azure region, but similar trends are seen for the other Azure regions as well. Fig. 8 shows the allocation imbalance when applying our placement heuristic. The figure shows the trade-off between deployment latency and allocation imbalance for different settings of the deflection threshold and deflection percentage. Increasing the deflection percentage for a given threshold results in better allocation balance (Fig. 8a) at the expense of higher request completion latency (Fig. 8b) and a larger load on the central controller.

It is noteworthy that for a given deflection percentage, lowering the deflection threshold, which would result in increasing the number of deflections, does not result in a proportionate decrease in allocation imbalance. For *e.g.,* with reference to Fig. 8a, one can see that there is a significant reduction in allocation imbalance between thresholds of 0.5 and 0.6. On the other hand, the reduction in allocation imbalance between 0.3 and 0.0 is much smaller. This result is intuitive since 0.5 threshold is close to the average resource commitment for the evaluated group of μDCs.

These results support the third hypothesis regarding the use of deflection for the latency/load-balance trade-off. Further, they suggest an intuitive policy for setting the deflection threshold, namely, the average resource commitment across equivalent μDCs. Such a simple policy would be easier to implement than striving for an optimal trade-off, which requires the knowledge of the lifetime of the deployed applications and may not be readily available.

6.3 End-to-end Evaluation

The microbenchmarks stress-tested the control planes with no real allocation of edge resources or execution of application components. In this subsection, we detail an E2E evaluation using the experimental platform and OneEdge's full implementation for running mockups of the exemplar situation awareness applications discussed in §3. The purpose of the evaluation is to verify our last hypothesis, namely, OneEdge's ability to provide both low latency for *S-App* applications and meet application SLOs expressed as latency bounds and spatial affinity (§3.2). Additionally, we study the effectiveness of the placement algorithm for reducing the load imbalance across latency-equivalent μDCs.

6.3.1 Experimental Setup

Applications. For end-to-end evaluations we experimented with two applications: **Drone** (based on [12] and [35]) as an instance of a *S-App* and **View-Fuse** (based on [34]) as an instance of a *C-App*.

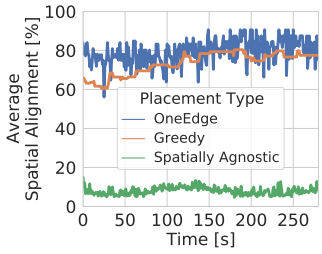
S-App uses a camera input and inertial measurement to determine the pose and location of the drone, using a Kalman-filter based algorithm. The drone application's pipeline comprises two stages: 1) feature tracking and detection from the inertial measurement unit (IMU) and cameras, and 2) pose state estimation (update). For our evaluations, we use a dataset generated using the ROS [25] framework for both the camera and inertial measurements [14]. To model the mobility of the drones (each drone is operating independently), we use the San Francisco cab dataset [24], associating an individual cab mobility to that of a drone. *S-App* is run with the above synthetic dataset and mobility data for a mockup of the *standalone* application for our evaluation studies.

C-App fuses the objects detected by multiple vehicles from their respective fields of view to create sub-regional view (Fig. 3), which is then sent back to the vehicles in the same geographical locality to improve collision avoidance decisions. To create a mock-up of this application for our evaluation purposes, we first created a dataset using Carla [10]. Specifically, we used 80+ cars plying through the most complex map available in Carla (called Town3). Carla simulator is run for 15 minutes and produces a spatio-temporal dataset consisting of object detections by individual vehicles. This dataset is then used as the input to *C-App* for a mock-up of the *coordinated* application for our evaluation studies.

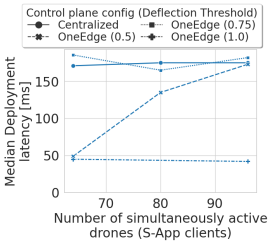
Mixed Workload Creation. For the E2E evaluation we wish to create a mixed workload consisting of both *standalone* and *coordinated* applications exercising the control plane simultaneously. The maps are different for the two applications in the above data collection. However, the only purpose for the map is to assign a spatial location for a client relative to others in the same application. Therefore, to unify the two map data, we shrunk the larger map (SF city) so that its four corners are aligned with the Carla Town3 map. The implication from the application point of view is that the drones appear to move slower than in the original dataset.

Control Plane Configurations: We consider 4 configurations: Centralized and OneEdge with three different deflection thresholds for *S-App* requests: 0.5, 0.75, and 1.0 (the deflection percentage is fixed at 100%).

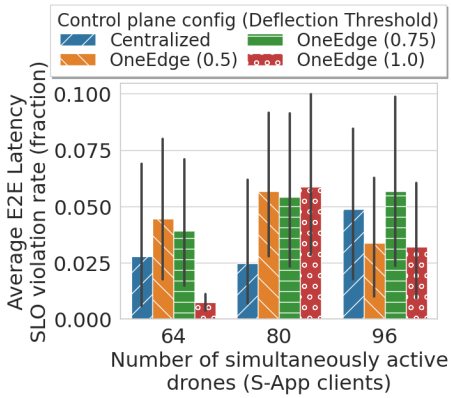
Evaluation Metrics: In addition to the control plane figures of merit such as deployment latency and spatial alignment,



(a) Spatial alignment for *C-App* application.



(b) Deployment latency for *S-App* application.



(c) E2E Latency SLO violations detected for *C-App* application.

Figure 9: Evaluation of E2E situation awareness applications.

we also record SLO violations for all deployed application components. For *S-App*, the latency bound is 12 ms for the first level (feature tracking) and 50 ms for the second level (update). For *C-App*, latency bounds are 10 ms for the first level (sub-region view) and 100 ms for the second level (region view), similar to prior work [16]. Any perceived latency exceeding these bounds is considered an SLO violation.

6.3.2 Analysis of Results

Fig. 9a shows the achieved spatial alignment for the *C-App* application. The graph depicts a representative window of time for the *C-App*'s execution in one metropolitan area. We partition the metropolitan area into 48 AoIs, calculate the spatial alignment (as per §3.2) for each AoI, and report the average spatial alignment achieved. An optimal placement would achieve 100% spatial alignment. We also plot a Greedy and a Spatially Agnostic placement, as reference points. Greedy selects the closest μDC every time a vehicle requests to connect to the View-Fuse *C-App*. Greedy is an idealized approximation of any real greedy implementation: it is computed offline, and does not account for migration or deployment latencies. Spatially Agnostic places each request on any μDC with available resources, without taking the client's location into account. Fig. 9a shows that OneEdge achieves a better overall spatial alignment than Greedy, with downward spikes attributed to the latency of migration causing the achieved spatial alignment to lag behind the ground truth of the vehicles' spatial affinity. The huge gap with Spatially Agnostic indicates the significance of spatial affinity in the control plane's placement decisions.

For Fig. 9b and Fig. 9c, we fix the clients using the *C-App* (vehicles) to 72 and sweep the number of clients using the *S-App* (drones). Fig. 9b shows the median deployment latency for *S-App* applications. When all deployments are handled locally—deflection threshold 1 and assuming sufficient μDC capacity—OneEdge's achieved deployment latency is more than 3× lower than centralized. As more requests are deflected, OneEdge's deployment latency converges with the centralized control plane's.

Finally, Fig. 9c shows the median fraction of SLO violations for the sub-region view application component of the *C-App* application under various control plane configurations. Similarly to Fig. 9b, the violations are presented with a varying number of active drones. All three OneEdge configurations display similar trends for SLO violation rates as centralized, while recording much better deployment latency for *S-App* applications. As expected, a higher deflection threshold in OneEdge yields lower *S-App* deployment latency. More Drones increase the probability of reaching a site's deflection threshold, leading to more deflections and thus higher latencies for *S-App* applications.

7 Conclusions and Future Work

OneEdge is an agile control plane for supporting situation awareness applications on geo-distributed edge infrastructures. OneEdge's contributions include its rich feature set and novel distributed state management that allows concurrent scheduling decisions at the edge sites and the central controller. The system has been evaluated with microbenchmarks and mock-up of situation awareness applications in a multi-region Azure setup.

Avenues for future work include (a) *Control plane federation*: We plan to partition the central controller into multiple such controllers with overlapping coverage regions, using a coordination mechanism akin to enhanced 2PC. Additionally, pushing some of the bookkeeping from the central controller to the edge sites (*e.g.*, deflection management) could further improve the system's scalability. (b) *Resource scheduler parallelization*: Currently, the central controller processes requests sequentially; we plan to apply datacenter scheduler optimizations [26] for parallelizing the central controller.

References

[1] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. 2017. Real-time video analytics: The killer app for edge computing. *computer* 50, 10 (2017), 58–67.

[2] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI*

14). 285–300.

[3] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 167–167.

[4] City-Data. (accessed May, 2021). FCC Registered Cell Phone and Antenna Towers in San Francisco, California. https://www.city-data.com/towers/cell-San-Francisco-California.html.

[5] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. 2019. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 177–192.

[6] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 499–510.

[7] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 97–110.

[8] Docker. 2020. Docker Engine overview. https://docs.docker.com/engine/.

[9] Docker. (accessed May, 2021). Runtime options with Memory, CPUs, and GPUs. https://docs.docker.com/config/containers/resource_constraints/.

[10] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[11] M. Gerla, E. Lee, G. Pau, and U. Lee. 2014. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*. 241–246.

[12] Samira Hayat, Roland Jung, Hermann Hellwagner, Christian Bettstetter, Driton Emini, and Dominik Schnieders. 2021. Edge Computing in 5G for Drone Navigation: What to Offload? *IEEE Robotics and Automation Letters* 6, 2 (2021), 2571–2578. https://doi.org/10.1109/LRA.2021.3062319

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*. 22–22.

[14] Roland Jung, Gernot Rischner, Eren Allak, Alexander Hardt-Stremayr, and Stephan Weiss. 2020. *AAU synthetic ROS dataset for VIO*. https://doi.org/10.5281/zenodo.3870851 The three datasets consists of 7 archives in total. The 1280x720 archive is was split in 2000 MB parts. To restore original archive please run: 'cat AAU_VIO_1280x960.zip.part* > AAU_VIO_1280x960.zip', then decompress it with capable tool.

[15] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 485–497.

[16] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 751–766. https://doi.org/10.1145/3173162.3173191

[17] Patrick Lindemann, Tae-Young Lee, and Gerhard Rigoll. 2018. Supporting Driver Situation Awareness for Autonomous Urban Driving with an Augmented-Reality Windshield Display. In *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*. IEEE, 358–363.

[18] Karim Manaouil and Adrien Lebre. 2020. *Kubernetes and the Edge?* Ph.D. Dissertation. Inria Rennes-Bretagne Atlantique.

[19] Linux Manual. (accessed May, 2021). Tc - traffic control. https://linux.die.net/man/8/tc.

[20] Microsoft Azure. (accessed May, 2021). Azure Geographies. https://azure.microsoft.com/en-us/global-infrastructure/geographies/.

[21] Walter Milliken, Trevor Mendez, and Dr. Craig Partridge. 1993. Host Anycasting Service. RFC 1546. https://doi.org/10.17487/RFC1546

[22] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[23] Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. 2019. Democratizing the network edge. *ACM SIGCOMM Computer Communication Review* 49, 2 (2019), 31–36.

[24] Michal Piorkowski, Natasa Sarafijanovic-Djukic, and Matthias Grossglauser. 2009. CRAWDAD data set epfl/mobility (v. 2009-02-24).

[25] ROS [n.d.]. Robot Operating System(ROS). https://www.ros.org/about-ros/.

[26] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 351–364.

[27] VaporIO (accessed May, 2021). VaporIO: a nationwide platform for edge. https://www.vapor.io.

[28] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[29] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[30] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. 2019. Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM Symposium on Cloud Computing*. 246–258.

[31] Wikipedia. (accessed May, 2021). Pokemon Go. https://en.wikipedia.org/wiki/Pok%C3%A9mon_Go.

[32] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. 2018. Extend Cloud to Edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 373–377.

[33] Zhuangdi Xu, Harshil Shah, and Umakishore Ramachandran. 2020. Coral-Pie: A Geo-Distributed Edge-compute Solution for Space-Time Vehicle Tracking. In *ACM/IFIP Middleware '20* (December, 2020).

[34] Z. Zhang, S. Wang, Y. Hong, L. Zhou, and Q. Hao. 2021. Distributed dynamic map fusion via federated learning for intelligent networked vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*.

[35] Alex Zihao Zhu, Nikolay Atanasov, and Kostas Daniilidis. 2017. Event-based visual inertial odometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5391–5399.