Vertex: A Unified Edge Computing Environment

Michael Perry
Scott Shenker, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-71
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-71.html

May 13, 2021

Vertex: A Unified Edge Computing Environment

Michael Perry

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Scott Shenker
Research Advisor

5/13/2021

(Date)

* * * * * * *

Professor Aurojit Panda
Second Reader

May 13, 2021

(Date)

Vertex: A Unified Edge Computing Environment

Michael Perry
UC Berkeley

Abstract

Edge computing, the notion of a machine or a cluster of machines that communicates with nearby clients in physically closer proximity than datacenters, has seen a constant increase in practical use in recent years. By expanding the computational power of the cloud while significantly reducing latency and alleviating bandwidth loads off datacenters, edge computing led to a new programming paradigm allowing new types of applications to emerge with clear benefits to end users. Despite these benefits, application development for the edge has not been as widespread as initially expected. Current edge offerings often fail to meet the requirements of the distributed, dynamic environment applications need to leverage the edge, as they neglect to address the high mobility and interactivity of devices. In this paper, we describe the new landscape of edge applications and their requirements, present core abstractions for ephemeral and long-living communication between entities, and propose Vertex, a unified environment that will profoundly accommodate edge applications while efficiently utilizing the geo-distributed nature of the edge.

1 Introduction

Exponential advancements and new technological innovations over the past several decades have changed the way we interact with computers, smartphones, sensors, and other devices. Cloud computing, since its emergence in 2005, was a necessary paradigm shift from the old view of single tenant servers to a new set of abstractions that allow decoupling resources from physical machines. The cloud can now host thousands of services with an almost unlimited compute power [6]. The onset of cloud computing coupled with an increase in computation power led to an exponential growth in the amount of data produced and processed, and to a demand for fast, high quality communication between systems. Autonomous cars, for example, are expected to generate about 40 TB of data in 8 hours of operation - 1.5 GB per second [36]. The amount of data to be processed, however, is not the only factor of essence. Though the cloud has been an efficient way for data processing due to its vast resources, the bandwidth of network and the propagation delay have become a bottleneck for computation-expensive real-time applications.

Edge computing geographically broadens the compute domain from datacenters to the edge of the network. The term *edge* signifies the *network edge*; any computing and network resources along the path between data sources and cloud datacenters [13], such as micro datacenters [32], IoT gateways [40], or even a single server within a few miles from the user. It is intended to make nearby compute resources available to clients by offering a large pool of resources over local machines with minimal latency, while reducing load from datacenters and eliminating the bandwidth bottleneck. These characteristics endorse diverse types of applications (which we later define as *edge applications* in 2.1) to focus their efforts on utilizing the edge beyond merely serving serving web content, such as fog robotics [19], video streaming [23], smart glasses [22], and web security [48].

Current offerings of edge providers focus on two main approaches. The first is providing developers with a low-level interface, such as virtual machines or containers, similar to a cloud environment [4]. While this provide ultimate flexibility by granting maximum control over resources and primitives, it could lead to high overheads as developers must handle low level building blocks, such as choosing and implementing their own primitives. This approach also limits edge providers as they are limited to only support VM/container abstractions and are limited in their fine-grained control over resources. The second approach is to offer a high-level interface for edge development, abstracting away operating concerns. Development is effectively easier since the interface is predetermined and eliminates the necessity for resource management, specialized security mechanisms, instances coordination, etc. The underlying mechanisms are left for the edge providers to manage, and thus as long as the interface is supported, the low-level infrastructure can be modified to allow innovation, optimizations, resource refinements and other technological improvements. Though this approach is more common with

large edge providers, it is often incomplete and lacks generality. Oftentimes, the interfaces are either application-specific or provide only a narrow set of primitives.

In the today's rapidly changing development environment, *edge applications* require large computation power, low latency for real-time coordination, and high throughput to accommodate the large data generation by end-users. At the same time, design and implementation complexities make simplified abstractions much more compelling, as seen by the vast adoption of serverless computing [46] for its alluring elasticity and generality.

Serverless computing alone, however, is missing key primitives for distributed computation and coordination. We identify the requirements for efficient distributed computation at the edge as twofold: a generalized, unified framework, accommodating all edge applications, and high-level abstractions for ephemeral communication and shared data, vastly alleviating implementation complexities for application developers. In this paper, we propose Vertex, a simple-to-use, high-level unified edge framework that abstracts away both ephemeral and long-living coordination using simple APIs for communication and data sharing, is generalized for all types of edge applications, and is particularly designed for the edge to satisfy performance requirements, fault tolerance and clients mobility.

The remainder of this paper is structured as follows:

- In §2, we discuss the significance of edge computing (§2.2), list the properties of the edge (§2.3) and the edge applications core requirements that the underlying system must address (§2.4.1), and demonstrate how the current ecosystem is lacking support to these concerns (§2.4.2).
- In §3, we present the high-level design of Vertex, and dive into its programming model and fit to edge applications.
- We then introduce the shared data subsystem in §4 and the communication component in §5, and discuss the requirements, implementation and interface of our newly proposed unified primitives.
- In §6, we present an overall evaluation of Vertex by showing performance measurements and latency benchmarks to verify Vertex's ability to change the current edge paradigm and estimate its performance in real-life scenarios.

2 Motivation

To capture the necessity of a unified environment for edge computing, we wish to demonstrate the essence of edge computing, describe its properties in our changing technological reality, and then discuss the limitations of current infrastructures as an evidence for Vertex's importance and urgency.

2.1 Definitions

To eliminate ambiguities, we first lay out terminology for clarity. Throughout this paper we will adhere to these definitions, albeit seldomly articulated differently in literature.

Edge Applications are user-facing applications (either directly or indirectly) that possesses certain properties that might not be attainable with cloud infrastructure alone, such as high mobility, extremely low latency or real-time aggregation.

Edge clients are end devices that utilize applications deployed at the edge, such as mobile phones, autonomous cars, sensors and the like.

Edge nodes are logical units of edge infrastructure over which applications can be deployed, such as servers or a cluster of machines at the edge of the network.

Backend nodes are servers in cloud datacenters. Edge applications may leverage the cloud as a backend, e.g. if an *edge node* only provides a portion of the functionality, a *backend node* can complete the remainder.

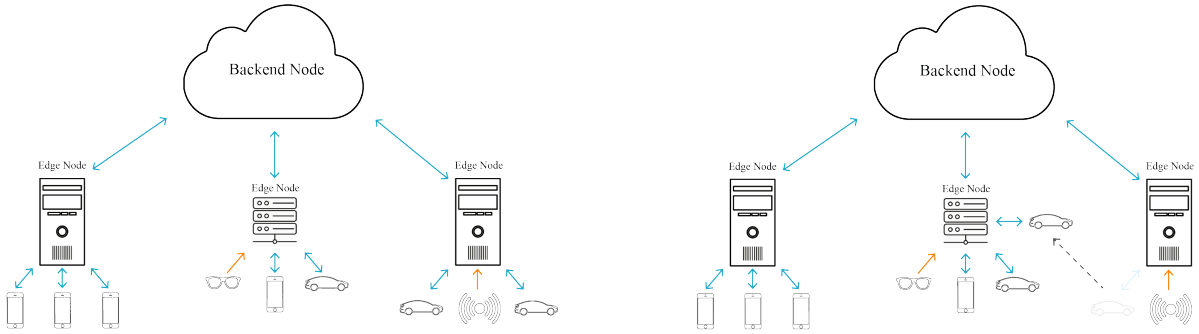
Edge providers are companies that sustain, manage and provide access to *edge nodes* for application developers.

2.2 The Significance of Edge Computing

Datacenters nowadays are abundant of computational resources that give an illusion of nearly infinite amount of processing power, subject only to the careful provisioning by the provider. But computational resources alone, albeit the existence of extremely efficient topology planning in datacenters [3], carefully designed network-level protocols [5], the use of RDMA to speed-up flows [12] and high bandwidth links inside and outside datacenters, is not sufficient for many real-time applications.

The main deficiency of cloud computing arises when considering real-time, latency sensitive applications. Autonomous cars, graphical mobile games, IoT applications and similar real-time applications rely on accurate delivery of information within strict requirements. In some cases, failure to meet these requirements could invalidate the use of the application altogeher. Such strict service-level agreements (SLA) between a provider to an application developer cannot be guaranteed for ultra low latency due to the extremely large, unexpected nature of network traffic within a datacenter. In a similar comparison, when accounting for propagation delay, the minimal user-datacenter latency would be much higher than that of a user-edge latency due to the physical proximity of an edge node to the client. In practice, Edge network latency can reach fractions of a datacenter latency [30].

In addition to strict latency concerns, datacenter bandwidth capacity might fail to meet the requirements of computation



(a) Snapshot of edge topology

(b) Edge topology after client migration

Figure 1: Edge nodes and edge clients exchange pre and post-processed data

heavy applications. Many datacenter networks are oversubscribed, sometimes as high as 40:1 in some Facebook datecenters [9], causing intra-datacenter traffic to contend for core bandwidth. As a result, datacenter bandwidth is unable to level with surging traffic requirements. While bandwidth capacity is gradually increasing and datacenter deployments are widening, the amount of data streamlined in for processing is scaling up accordingly, leaving throughput concerns virtually unattended at the datacenter level. Offloading network traffic to the edges of the network will not only support agile data-heavy computations, but will also alleviate pressure of cloud providers and allow smooth core traffic processing for other cloud uses.

Another use case for the edge is expensive, short-living computations. These types of user-initiated requests require processing power that extends beyond end devices (i.e. mobile devices, personal computers, etc) but doesn't logically require any attention from the datacenter or the cloud. Such applications traditionally lie in the intersection between machine learning and signal processing, commonly referred to as **Federated Learning (FL)**. In Federated Learning, end devices use their local data to collaboratively train a ML model. The end devices then send the model updates rather than raw data to the server for aggregation [25]. FL has found recent success in several application, such as Google's Gboard's *FedAvg* algorithm [21], or in predictive models for diagnosis in health AI [18].

2.3 Edge Properties for Vertex

Certain types of edge nodes are specialized and meant to fulfill a single need or provide an application-specific support. Offerings such as AWS Outposts [33], for example, could serve as native extensions to Amazon cloud services running on the company's on-premise infrastructure. In this paper, we intend to focus on generalized purpose multitenant edge nodes that are capable of supporting number of applications and can be shared among unrelated edge clients to serve multiple edge applications at once. We speculate that the multitenant model will shape the edge as a natural extension to the cloud. Unleashing the advantages of the edge (2.2) requires another base assumption regarding the physical proximity to edge clients. Physical proximity affects end-to-end latency, economically viable bandwidth, establishment of trust, and survivability. It also proves efficient in masking cloud outages by dishing a fallback service that can serve local clients temporarily while cloud services are unresponsive [16]. Our assumed core benefits of the edge could only be achieved if edge nodes are assumed to be abundant and spatially sparse at the edge of the networks (although limited in their scope of service per instance), and within close proximity to the edge clients. This premise is based upon existing evidence of extensive edge infrastructure by several providers [7], [14].

As for edge clients, we target mobile devices that frequently change physical locations, such as smartphones or autonomous cars. By default, edge clients will be assigned to the nearest edge, and could request to connect to another, closer edge as they change their physical location. We claim that ultimately, in the current landscape of application development and the increasing computation capabilities of personal devices, the purpose of edge nodes will mainly be to serve many such clients and cater to their high mobility with appropriate primitives. Such model, we believe, would be indispensable to the future of mobile applications and to the benefit of new technologies.

The edge infrastructure possesses an additional key trait regarding its failure modes. As resources are naturally more scarce at the edge, it would be relatively more prone to failures with less fault-tolerance mechanisms than the cloud, and weaker reliability guarantees. Since we model our edge ecosystem with a backend node within arms reach at all times, those weaker guarantees could be masked by live communi-

cation and localized edge recovery systems.

Figure 1 demonstrates our edge model. The color of arrows represent the type of edge application, and the direction of the arrows represent the flow of computation. The blue application could represent, for example, a CDN-based application such as *Apple Music*, where edge nodes respond to requests from the mobile phones and serve the client with cached entries, improving response latency compared to direct user-cloud communication. The orange application might represent a video application, where the edge node receives a visual feed from the IoT glasses and compresses the data before sending it to the cloud for further processing, reducing the amount of network traffic. The difference between 1a and 1b demonstrates our mobility assumptions. The autonomous car moves physically closer from the rightmost edge node to the middle edge node and can now communicates with the middle edge node, as denoted by the dotted arrow.

2.4 The Expected vs. The Reality

To recognize the benefits of our proposed unified environment for edge computing, we first explore the requirements of edge applications and the limitations of current offerings by edge or cloud providers. We further build upon these ubiquitous demands to define key building blocks that our system design must support, and materialize our observations into one simplified environment with flexible primitives.

2.4.1 Requirements For Edge Applications

In academia, there has been extensive research attempting to identify the needs of edge applications and the ideal underlying system to accommodate those demands [29]. Articulation of these needs, however, remains somewhat obscure due to the diversity of applications that stand to benefit from the edge. Some applications emphasize the distributed nature of the edge and require stringent latency needs and coordination across distributed clients [17]. On the other hand, Federated Learning applications [25] leverage the edge as a mean of computational offloading, and thus ignore the need for inter-edge or inter-client coordination. We therefore identify several key features for edge applications:

Migration: As stated in 2.3, edge clients are assumed to be assigned to the closest edge node to maximize application utility. Based on the assumption that existing mechanisms can be employed for clients to learn about nearby edge nodes, as a mean to efficiently handle clients' mobility while preserving state stored on the edge, applications should have the ability to rapidly migrate state from one edge node to another. For instance, smart cars [24] working together to form an interactive map of other cars on the road may desire to migrate to the next edge node if they have traveled substantially far away from the first edge node and are now in closer proximity to a different edge node.

Location-agnostic communication: The dynamic edge environment is characterized by abundant edge clients and edge nodes who do not know about the existence or location of other entities. Since the application logic is potentially spread out across edge clients, edge nodes and backend nodes, communication between distributed components of an application may be difficult. The underlying edge system must provide primitives for distributed application components to communicate regardless of location.

Consistent shared data: Many applications require working on shared resources (such as datasets or collaborative objects), which makes consistent access to data mandatory. A single consistency guarantee might not suffice to the variety of edge application, thus a range of consistency models must be supplied to match application needs - strict consistency levels provide guarantees for shared data access but incur large overheads, while weak consistency levels compromise on some data inconsistencies but provide reduced latency. Strict consistency requirements may include online gaming [17] as players expect their adversaries to constantly remain up-to-date, while weaker consistency should be suffice for Federated Learning [25] and other machine learning models.

Fault tolerance: As edge nodes are intended to store state shared between edge clients and other edge and backend nodes, they must provide guarantees about failures and recovery. As in any system with a distributed nature failures will be imminent, and the underlying system must provide well-defined behavior for mitigating failures [11].

Low latency and high bandwidth: Since one of the core advantages of the edge is close proximity to the client and reduced latency, the underlying system must be efficient in terms of response latency. This leads to an emphasis in two planes: the startup overhead of the serverless environment should be extremely low, and the edge environment must ensure that rich functionality does not incur unnecessary latency or delays. In addition, though not in the scope of our work, the edge node must guarantee sufficient bandwidth, especially for the benefit of applications that require frequent heavy data transfer.

2.4.2 Limitations of Current Offerings

Given these set of core functionalities that edge systems must provide, we claim that current edge offerings are insufficient and lacking support that would allow applications to leverage the edge. We distinguish research in academia to industry providers: recent academic papers emphasize low-level interfaces and mainly focus on application-specific environments,

while industry offerings are incomplete, lack generalized features to accommodate the diverse nature of edge applications, and miss core primitives.

Academia: Academic research in edge computing over the past few years have been diverse, addressing specific challenges of programming for the edge, and ranging from consistency models for shared data abstractions [11], lightweight function-as-a-service platforms for edge nodes for faster execution [28], or communication mechanisms for a specific use-case in autonomous cars [26]. Consistently, however, the main discussion points fail to reflect all necessary primitives for generalized support in edge applications, while some still focus on low-level interfaces rather than development-centered abstractions [27].

Industry: Different edge providers exhibit varied interfaces to their programming environment, leading to an inconsistent developer experience. Small number of providers lay out low-level interfaces like VMs and containers (e.g. MobiledgeX [44]). These interfaces provide no primitives, leaving developers to manually handle distributed communication. Some of the most popular approaches consist of high-level interfaces for function-as-a-service (e.g. AWS Lambda@Edge [34]) or core abstractions like shared data [38], but all share two primary shortcomings with the offered primitives. First, these primitives are often incomplete due to lack of support for either ephemeral communication patterns (e.g. Microsoft Azure for the edge [43]), or for the shared data abstraction. Second, even for the primitives that are provided, the semantics vary across edge providers: AWS s3 support strict read-after-write consistency [31], while CloudFlare's Key-Value Store have eventual last-writer-wins consistency [38].

With industry endeavors to make edge computing more widespread and support new features, the vendor-specific underlying implementation intricacies lead to cross-provider utilization constraints. As the rapid adoption of multi-cloud solutions by corporations [10] and economic research [1] has implied, vendor lock-in is undesirable for the clients of cloud providers. Vendor lock-in also implies that clients may not be able to use compute resources at the physically closest edge node, which is necessary to unlock the benefits of using the edge in the first place. Additionally, as empirical research has found, while existing edge nodes generally provide better latency than cloud communication, it still incurs fairly high latency compared to the desired use cases of edge applications. Lambda@Edge, for example, is above 150 ms latency on average [30], though edge technologies could possibly incur latency as low as 20 ms [45].

Given this ecosystem, we make the observation that edge offerings lack a uniform and extensive approach that holistically deals with migration, communication, and consistent and fault-tolerant access to shared data. Without such approach, customers are constrained to use the services of a particular edge provider either due to vendor lock-in or due to inconsistent semantics across offerings, which would impose a barrier for developers to efficiently migrate customers to the nearest edge and, at the macro level, would make writing edge applications unnecessarily difficult, potentially deterring developers.

To combat these limitations, we present Vertex, a unified edge environment that supports the serverless model, offers high-level interface which reduces development complexities and allows innovation at the underlying system level, and is rich in primitives that abstracts away ephemeral and long-living data sharing from developers. We envision that Vertex can be adopted across all edge providers and help unlock the potential of the edge for diverse applications.

3 Design

Vertex is an edge system that runs across edge clients, edge nodes, and backed nodes. Figure 2 demonstrates the system from an overview, which will be broken into components in this section: in 3.1 we present the high-level abstractions (3.1.1) used to devise the system and the core primitives we chose to focus our efforts on (3.1.2), we then discuss the container management structure in 3.2 before briefly mentioning our fault-tolerance integration in 3.3.

3.1 Service Model

Adjusting to the edge programming paradigm and satisfying the comprehensive needs of edge applications, Vertex is built with edge application requirements in mind. Core operations are meticulously designed to satisfy the ultra low latency constraints, most taking less than a millisecond to perform. We accommodated edge developers' needs with high-level abstractions, while catering to edge providers with a flexible underlying implementation that easily allows innovation. Further, since mobile clients would be of major interest for edge development purposes, we emphasized seamless mobility between edge nodes for a continuous interaction with edge clients despite decreasing physical proximity. Lastly, we protected Vertex from failures by deploying a resilient edge-computing recovery mechanism.

3.1.1 High-level Abstractions

Vertex's central design decision is to abstract complexities of managing a virtual machine away from application developers to divert their attention to application intricacies from resource management. Though VMs offer many benefits to developers and providers, such as flexible resource allocation and priority management, we conjecture that implementing edge services based on virtual machines comes at a large price

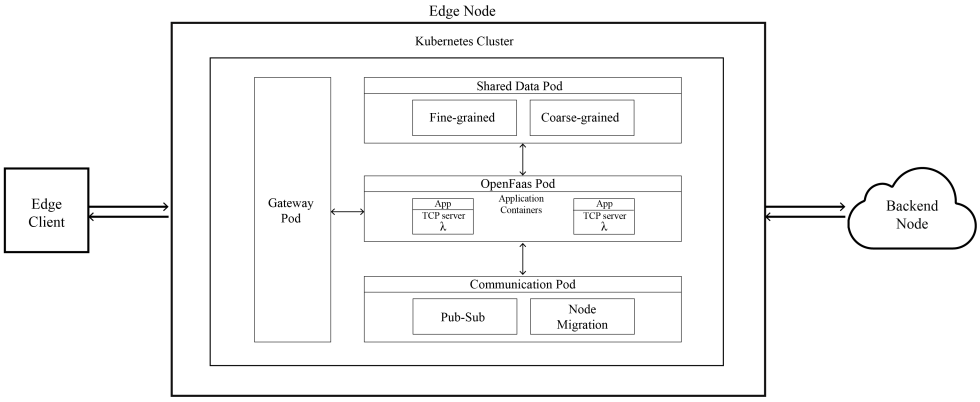


Figure 2: Vertex Design

which deems it insufficient for application development for the edge.

From a programmer's perspective, high-level abstractions are desired over an OS-like model. Application developers would prefer focusing their efforts on creating new features and improving the app's design and internal mechanisms over managing resources at the edge level. The lack of assumed primitives also require much more work at the systems level to create low-level building blocks that support coordination with various application instances over geo-distributed edge nodes.

On the providers' side, offering VMs taxes the physical machines in a large amount of RAM and CPU cycles, since each VM creates a virtual copy of all hardware that the operating system needs to run. VMs are based on a hypervisor that largely complicates introducing new technologies in the underlying system level for controlling resources below the operating system. Similarly, containers as the lowest-level abstraction will undermine providers' efforts to improve performance, to all of which the serverless paradigm grants an easy solution for. With less operating overhead and accurate resource provisioning, providers can also offer a more precise resource consumption based monetization.

3.1.2 Core Primitives

Identifying common needs for the diverse range of edge applications was the leading guideline for devising Vertex's subsystems. As mentioned in 2.4.1, the high-pace mobility of clients and the importance of geo-distributed coordination between entities resulted in identifying the core primitives for the edge.

Execution Environment: The benefits of cloud-based serverless computing, and especially the Function-as-a-service (FaaS) model, can extend to the edge. Vertex leverages both the performance benefits of FaaS, since invocations that spin up a compute environment must be fast, and the efficient hardware utilization through statistical multiplexing. Vertex uses *OpenFaas* [46] as the underlying execution environment, but implements a wrapper around it so that any serverless engine can be used instead. Applications developers can register their functions to Vertex and allow edge clients to trigger these functions at edge nodes, respectively. Invoked functions in the execution environment can communicate with other functions or instances using the two other Vertex primitives, communication and shared data, to share state among devices.

Communication: Instantaneous or short-living interactions between edge clients and edge nodes require ephemeral coordination between functions. The communication must be location-agnostic to account for the distributed nature of application instances and their frequent mobility, extremely lightweight to prevent congesting client devices with state updates, and must efficiently distribute state across relevant edge nodes (but not all). State filtering across edge nodes and clients occur based on topic, set by the application logic. Vertex adopts a Pub-Sub communication model for client-edge communication, reliable state migration between edge nodes, and a reliable node-backend coordination to establish a connection between all edge nodes for a specific topic, regardless of physical location.

Shared Data: Interactions requiring long-living state would utilize the shared data abstraction, an object-based library for shared reads and shared writes exposed to the serverless execution environment. Based on the scale of computation, the shared data subsystem offers two sub-primitives: fine-grained objects intended for short time-frame computations, and coarse-grained objects, purposed for long time-frame computations. As some applications have different priorities in terms of consistency levels and due to the inherent correlation between stricter consistency models and weaker performance guarantees, the shared data abstraction offers

different consistency levels: fine-grained objects could utilize linearizable and eventual consistencies, and coarse-grained data could utilize the strict serializable consistency model.

3.2 Deployment

Being a comprehensive edge framework that applications can be deployed over, Vertex runs across edge nodes and backend nodes. Edge nodes are organized as Kubernetes [42] clusters, and each node runs an instance of a serverless engine in a designated OpenFaas pod. Each application lives in a designated container inside the serverless pod along with other containerized micro-services and libraries that the application needs for operation, as shown in figure 2. In order to minimize latency and increase OpenFaas' edge compatibility, we modified OpenFaas' basic gateway to remove its control plane latency, which resulted in a significant reduction in runtime.

In addition to the OpenFaas pod, the communication and shared data subsystems also implement a microservice-based approach and are deployed on distinguished pods. The two components run server processes that accept client traffic via remote procedure calls (RPCs), though they directly interact with the execution environment solely, which forwards client requests accordingly. The RPC libraries are packed along with function code so they can invoke the API and support a set of interfaces implemented by client libraries. While the client libraries are automatically included on the application's container, its extensible structure enables support for multiple languages.

Kubernetes' convenient resource management, flexible control plane and containerized structure would make Vertex extremely appealing and easily adoptable for providers, though our mechanism is not tied to Kubernetes and could be replaced with any framework that supports composable units.

3.3 Fault-tolerance

Preventing state loss upon edge failures has additional complexities to the familiar client-server paradigm in which a stateful server provides services to multiple clients. The edge processor is logically located between the client and the server, which complicates maintaining consistency in the case of an edge failure due its scarce resources. Since the cloud does not store the edge state, deploying another edge will lose it state at the time of failure. To address these limitations, we deploy CESSNA [20], a low-level edge framework that supports local recovery of temporary, unshared state in case of failures.

4 Shared Data

Vertex's significant contributions begin with its shared data model, which revolves around varying consistency levels. Vertex's shared data model was initially designed to provide strong consistency (both linearizability and serializability) in a fault-tolerant manner since industrial offerings often omit these from their features. However, we elected to extend the consistency model to support eventual consistency to accommodate an additional range of robust use cases. The shared data subsystem is composed of a granular object-based library exposed to functions, which differentiats fine-grained, lightweight objects intended for short-timescale computation, and large, coarse-grained objects intended for long-timescale computation.

4.1 Requirements

The shared data subsystem accommodates applications' data sharing needs similarly to cloud computing, with one major difference: extremely low latency. Access to shared objects must be consistent as edge clients must reason about the validity of the information, and data must be replicated strategically to avoid corruption of data upon failures. With that in mind, as one of Vertex's main ideals is generality, the shared data component must be extensive enough so that applications can pick their preferred scope of data they handle at once. Our shared data abstraction is supported by two underlying approaches and can be leveraged when distributed edge nodes are operating on the same dataset. The first is a fine-grained approach which is designed for short, flexibly consistent operations. The second is a coarse-grained approach designed for long, strongly consistent, dynamic operations. Both of these approaches provide a way to handle fault tolerance and optimize performance, and both leverage a backend node to enforce consistency.

4.2 Fine-Grained

The fine-grained abstraction is designed for short-lived consistent operations on shared data. In simple words, fine-grained consistency can be summarized as bringing *compute to data* - Vertex checks whether the edge node or the backend node possesses control over the data, and performs the computation at the corresponding node. Given the short-lived nature of the operations there is a need to limit the amount of overhead per request. To reduce the raw overhead from concurrency management, we choose to support linearizability, which enables arbitrary interleaving of operations. Additionally, we added eventual consistency support for workloads that do not need strong consistency and wish to minimize overhead.

To further reduce operation overhead, we allow dynamic replication and placement of objects. Replication allows us to exploit the read/write ratio of a given application to improve performance while placement allows us to exploit locality. Ideally, the replication and placement decisions made by our shared data abstraction should be independent of the application and require no changes from the developer.

4.2.1 Linearizability

To provide linearizability in the face of object-level replication across edge nodes, Vertex leverages the MESI cache coherence protocol [8] and coordinates individual object access across edge nodes. Due to the low latency requirements of the edge, we made extensive performance optimizations that prevent excessive overhead of broadcasting shared objects updates across edge nodes. Vertex uses a directory-based write-invalidation protocol that invalidates replicas of that shared object upon a write. To support location-agnostic inter-edge coordination, this directory is maintained at two granularities - first, the backend node maintains a global directory which tracks shared objects coherence states across edge nodes, and second, each edge node maintains a local directory that keeps track of the coherence states of objects at that node.

To implement the MESI protocol, shared objects at a particular edge node can be in a *modified*, *exclusive*, *shared*, or *invalid* state. When the object is initialized, only the backend node has exclusive access to the shared object. Edge nodes can only write to an object if that object is in a modified or exclusive state at that node, but reads can be performed in those two states as well as the shared state. An object in a modified state implies that the edge node has not yet written back the value to the backend node.

The computational flow starts with an edge client triggering a function that makes one or more requests to read or write a shared object through **shared_read** and **shared_write**. The edge node first checks if it has access to the entire read and write set for that function. If it does, the execution environment performs the function and returns the result. Otherwise, the shared data pod communicates to the backend node, which then checks if it has read access to the entire read set and write access to the entire write set of shared objects in question. If it does not, it checks the write set of the requesting edge node and invalidates replicas at other edge nodes that have exclusive access to those objects. At this point, the backend node executes the function on behalf of the original requesting edge node, gives this edge node upgraded coherence states (exclusive for written objects and shared for read objects), and communicates this information back to the original edge node.

4.2.2 Eventual Consistency

The shared data fine-grained paradigm additionally supports eventual consistency, which enables application developers to write merge functions for shard objects along with a recency parameter indicating how much time can elapse before data can be assumed to be stale. Merge functions, such as last-writer-wins, allow applications to reserve conflicted versions of shared objects based on the application's semantics, and the recency parameter force timely synchronization of shared objects. Eventual consistency is a weaker consistency model than linearizability and provides weakened guarantees for the benefit of a much simpler coordination mechanism and much lower latency. Only certain type of functions (e.g. merge functions), however, can benefit from its merit.

4.2.3 Fault Tolerance

The shared data mechanism must ensure fault tolerance to withstand failures without losing essential information. To deal with failed edge nodes or network partitions, Vertex maintains a persistent copy of the objects at the backend node, which is assumed to be fault tolerant by other conventional methods. If the failed edge node had special access permissions to an object, the backend node receives those permissions to that object.

4.3 Coarse-Grained

Our coarse-grained abstraction, on the other hand, is designed for long-lived consistent operations on shared data. The long-lived nature of operations leaves a lower burden on reducing overheads, which ultimately lead us to support serializability as a strong consistency model. Compared to the fine-grained lightweight model of *compute to data*, our coarse-grained approach can be described as bringing *data to compute*, allowing functions to lock a set of objects and perform computations locally at edge nodes when triggered by edge clients. When a client invokes a function that makes a serializable operation request to the edge's shared data subsystem, the subsystem will contact the backend to lock the read and write set. The backend will give the edge subsystem access to the data, whenever possible, using its local reader/writer locks. The edge node then completes the operation and returns the result of the request to the client. To protect against edge failures, our coarse grained module uses leases instead of pure locks and acts as a cache over cloud blob storage.

4.3.1 Serializability

Due to the longer persistence of shared objects and potentially large amount of reads and writes, we found serializability to be a more fitting mechanism for enforcing strong consistency. In particular, Vertex's serializability mechanism enforces snapshot isolation [2], which provides most of the consistency guarantees of serializabilty while also improving performance. Vertex assigns globally unique logical start times to functions invocations. Throughout its lifetime, a function may read shared objects (i.e. using **shared_read**) if no other function with a logically later start time has written to those objects. Otherwise, the function aborts and is restarted. If the function has to write shared objects (i.e. using **shared_write**), Vertex buffers these writes, and these writes only persist if the function is able to commit after it completes its execution.

A function can commit if two conditions are met: the function must be the only one writing to the object, and no other function that has a later logical start time has read the objects that the original function has written. To verify these conditions, Vertex leverages per-object reader-writer locks so that functions can ensure they have write access to objects. If this lock cannot be acquired, Vertex will abort and restart this function.

Vertex ensures that writes are atomic, i.e. either all or none of a function's writes become available to other functions, to provide stronger consistency guarantees and eliminate the risk of leaving data in an undesirable data upon aborting an operation or failing to complete a write. Vertex also caches objects at edge nodes to reduce overhead stemming from interacting with the backend node or other edge nodes using the shared object.

4.4 Interface and Implementation

For the application developer, making use of shared data requires little change, as functions will merely import the supplied shared data client library to make requests. Shared functions are the unit of execution on shared data, performed and made consistent by the shared data subsystem on behalf of the edge application. To enable a function to be executed as a *shared_fn*, the developer will annotate it with the object read/write sets for the function as a function comment. Vertex's application parser reads these annotations and stores the read/write sets for each function transparently from the developer. Given that a *shared_fn* call may be migrated arbitrarily, the developer is responsible for writing the function such that each *shared_fn* can be singly executed by invoking the function using the operation name. This will require reading and writing shared objects using *shared_read* and *shared_write* which automatically serialize and deserialize native language-specific objects.

Both fine-grained and coarse-grained abstractions run as independent processes in the shared data pod (as depicted in figure 2). Application containers in the execution environment can access a designated shared data client to make *shared_fn* requests, while invocations sent to the same container utilize the same shared data client to reduce additional instance overhead. To conform to universal shared communication conventions, each shared object, be it fine-grained or coarse-grained, implements the following abstraction: **Read**, **Write**, **Lock**, and **Unlock**. Upon shared function request, the shared data client invokes the function, reads stored maps to determine which objects need to be altered or read and the specified consistency levels by the client. The shared data client then locks the object, read or write data as necessary, and then unlocks the object and returns the function result.

5 Communication

While shared data is intended to maintain persistent state, applications may also require ephemeral communication across edge clients and edge nodes. The communication primitive must be location-agnostic and account for edge client mobility, as edge clients that connect to another edge node due to physical proximity must adjust quickly without losing prior state or missing new incoming state updates. To achieve this, Vertex includes a hefty wrapper for a pub-sub interface, accompanied by a primitive for migrating communication state. Under the hood, edge nodes frequently communicate with the backend node to publish messages to other edge nodes, while all state updates are efficiently stored at the edge until explicitly requested from edge clients. In this section we will interchangeably use *state updates* and *messages* to convey the rudimentary elements passed via this subsystem.

5.1 Requirements

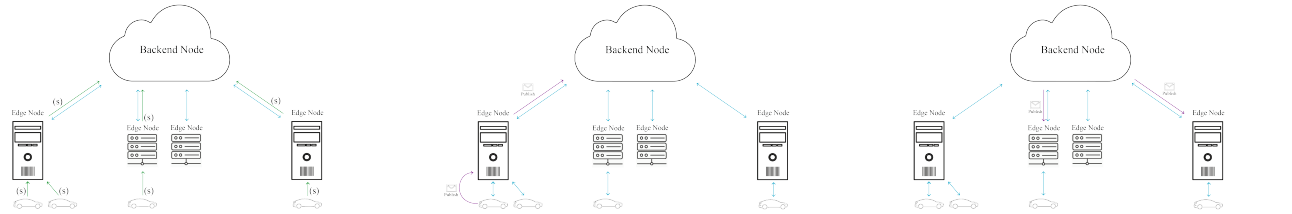
The communication subsystem was carefully designed to obey high performance standards due to its messages' ephemeral and agile nature of use. We emphasized 3 main characteristics when designing the communication primitive: location-independent coordination, low latency and minimal operational overhead, and support for mobile clients.

5.1.1 Location-Agnostic Patterns

All state updates passed through this module must contain information relevant to a specific edge client and may, or may not, additionally include details about their physical location (e.g. in the case of autonomous cars or location-based multiplayer games, like Pokemon Go [15]). However, the edge client's physical location, or that of other instances demanding state updates, should not be a factor in distributing state updates. Messages should be delivered to all relevant clients regardless of any actor's physical location. The *topic* subscription primitive of the Pub-Sub model 5.2.1 aligns with this goal perfectly, as edge clients can receive messages based on a shared topic regardless of location. Edge nodes leverage continuous connection to the backend node to distribute messages to all other edge nodes and edge clients.

5.1.2 High Efficiency and Minimal Overhead

Intended for time-sensitive coordination, the communication subsystem must be extremely efficient handling message delivery to other edge nodes or to edge clients, incurring minimal overhead. We implemented communication in C++ and built it around ZeroMQ sockets to leverage their low latency and support for atomic multipart messages. Our design emphasized minimal busy-waiting and low input-dependent iterations to reduce variability in duration of command executions.



(a) Edge clients subscribe to a topic

(b) Edge client publishes a message. Message is stored on edge and delivered to backend node

(c) Backend node publishes message to all subscribed edge nodes. Updates are stored on the edge nodes

Figure 3: Communication Publish-Subscribe Model

5.1.3 Support Mobility

Based on our hypothesis conjecturing large number of mobile edge clients, short-living coordination must support **migration** (2.4.1). After establishing close proximity to a new edge node, from the communication component perspective, migration is defined as the act of transferring all app-related state updates for a specific client from one edge node to another with minimal latency, while avoiding the loss of messages during the transfer period.

5.2 Communication Models

5.2.1 Publish-Subscribe Model

The communication subsystem's main coordination model is the *Publish-Subscribe pattern*, as shown in Figure 3. In pub-sub, each endpoint, i.e. an edge client or an edge node, first **subscribe** to a certain *topic* - a unique trait or title which all related messages share - they wish to follow (3a). Subscribing to a specific topic allows entities to send state updates, and to receive all state updates published anytime after subscription. Senders *publish* messages to the edge node (3b), which stores the messages locally. To deliver messages to all subscribed edge clients including those connected to geographically remote nodes, the edge node reliably sends the published message to the backend node. The backend node acknowledges the received message, and then broadcasts the message to all subscribed edge nodes for temporary local storage (3c).

The pub-sub model provides an efficient solution to the edge environment requirements. By leveraging the edge and backend nodes as middlemen, we decouple sending messages from receiving: every edge node store received messages and a corresponding offset for each topic, while storing the last offset queried by each client. Edge clients can therefore publish messages quickly due to the distributed responsibility for message delivery, while receiving pending messages immediately upon request. Unique topic titles permit applications to use application-specific naming while integrating a simple single-to-many delivery convention.

We observe that this mechanism achieves location-agnostic communication in two granularities. From the edge client perspective, they do not need to know the locations of other edge clients *a priori* to sending or receiving messages. From the edge node perspective, the set of edge nodes needed for an application is also not known *a priori* since this set depends on mobile edge clients dynamically attaching to local edge nodes. Vertex leverages the backend node as a rendezvous point as edge nodes can publish and subscribe to topics without any prior knowledge as to where is the publishing client or node. In our current prototype, the backend node does not store any state. Depending on the desired lifetime span of messages, vertex can store crossing messages on the backend node as a recovery mechanism for the communication subsystem in case of an edge failure.

5.2.2 Inter-Edge Data Transfer

Our assumptions of client mobility in 2.4.1 raises the question of what happens to the communication state if a mobile edge client reconnects to a closer edge node. More specifically, if the client migrates edge nodes, the set of topics it is subscribed to as well as its message offsets are stored at its previous edge node. Prior to any edge node - edge node interaction, we have no convenant that the new edge even knows about the existence of these topics or contains messages that the mobile client has yet to pull. To address this, Vertex supports client-driven mobility enabling clients to **migrate** their communication state to the new, closer edge node. Internally, upon a client's migrate call from edge node A to edge node B, edge node A will send the set of all topics and unrequested messages for that client, to edge node B. Edge node A will remove the migrated state only after receiving a receipt acknowledgment from edge node B, ensuring reliability and eliminating the risk of losing information. After migration, node B can take over in servicing the client for communication.

5.2.3 Reliable Delivery

As we concluded in 2, edge applications are often data-heavy, agile and require efficient coordination between geographically remote devices. Since communicating without delivery guarantees may yield failure to comply with some of the basic needs of edge applications, we built Vertex to be a reliable medium for state distribution with strict guarantees on successful message delivery.

Edge node-client message delivery is guaranteed by design; edge clients request messages only when they are ready to receive them, and the edge node will not dispose of these messages until the client has received all their pending messages atomically. The communication subsystem serializes requests and responses using Google's Protobuf [37] and employs gRPC [41] for client-node interaction, which alerts upon unsuccessful delivery of requests or responses.

At the lowest-level, inter-edge and edge-backend communication is done using ZeroMQ's reliable delivery [49]. In a nutshell, ZeroMQ's reliable sockets (*request-reply pattern*) atomically send messages to the receiver, and then repeatedly query them for an acknowledgement. The acknowledgement message contains some part of the original message, to validate the correctness of data received at the other end. If no acknowledgement was received after a set period of attempts (3) sent within a set time interval (1 second per attempt), the sender can confirm the unsuccessful delivery of the message, and react accordingly.

5.3 Implementation

In this section we will discuss the communication subsystem's implementation details and ration about decisions we have made and technologies we elected to use to achieve our ephemeral coordination goals.

5.3.1 Internal and External Coordination

Traditional communication methods involve two or more direct players: one acts as the sender, with one or more recipients. While this applies to the classic client-server or peer-to-peer models, the edge introduces a new environment that imposes intricacies embodied as *indirect entities* (i.e. the edge node and the backend node; neither of whom produce messages), which demand a significantly more comprehensive set of internal and external interactions.

Edge Client - Edge Node: The edge client establishes connection to the edge node by running a Python-based client library, specifying the nodes IP address. Instantiating the communication instance allows direct interaction with the communication module using its core API (5.4). All endpoints are exposed using gRPC [41] on both entities, i.e. an edge client publishing a state update will invoke a gRPC call to the connected edge, which will change its internal state accordingly, and an API call to get_messages, will deliver all state for some topic to the client over Protocol Buffers do be deserialized at the client's process.

Edge Node - Edge Node: Edge nodes communicate with each other per clients' request for migration. The edge client will specify the address of the new edge node they want to connect to, and the two edge nodes will reliably transfer topics and related state (5.2.3). Each edge node supporting vertex has a set of two ZeroMQ [49] *request-reply* sockets to carry out migration operations: a *request* socket initiates migration from edge node A to edge node B by stacking multipart messages containing user ID, list of topics, and a message stream for each topic. The *reply* socket receives the migrated state on edge node B, puts it in corresponding message queues (5.3.2), and sends and acknowledgement message if migration was successful. Edge node A, in turn, unsubscribes the migrated edge client from all relevant topics, and clears out irrelevant remaining state.

Edge Node - Backend Node: Lastly, to fully accomplish location-agnostic coordination, edge nodes automatically subscribe to the backend node to each topic that their connected edge clients subscribe to. Again utilizing the *request-reply pattern*, a *request* socket is initialized on every edge node with one purpose: send every single message that its edge client published, to the backend node. The backend node initializes a *reply* socket, and acknowledges the successful receipt of a published message. Following that, the backend node uses a *publish* socket to broadcast the message to every subscribed edge node in the topology. Edge nodes receive the message using a *subscribe* socket and store the messages in the appropriate message queue.

5.3.2 Message Queues

One of the main differentiators of our communication design from other pub-sub models is **where** pending messages are stored. Most common pub-sub designs avoid temporary state storage on the local machine, and as we envisioned the edge carrying the load of storing state before distributing to clients upon request, we incorporated message queues to prevent the native pub-sub design from directly offloading messages to clients.

Our pub-sub model incorporates an in-house thread-safe message queue that stores only one copy of any incoming state update per topic regardless of the number of subscribers, and a corresponding offset. The message queue keeps track of the number of subscribers at all times, and optimizes the queue (i.e. removes old messages and update offset numbering) during operational downtimes.

5.4 Interface and Core API

When devising the communication interface, we set two basic guidelines: generality and simplicity. Generality is required to provide support to all various types of edge applications deployed on the edge, and we emphasized a simple interface identical to the standard pub-sub model, to ease user interaction with the familiar interface and to enhance the usability of the subsystem as a whole.

5.4.1 Subscribe

Subscribing is the act of requesting the edge node (or the backend node, as depicted in 5.3.1) on their behalf (i.e. the edge client or edge node, respectively) to store message updates for a specific topic temporarily. A subscribed client is guaranteed to receive all state updates from all other edge clients and nodes (through the backend node) about a specific topic. In contrast, **unsubscribe** is the act of delisting oneself from receiving any further updates about a topic, de-facto removing all interest in the topic. By design, unsubscribing erases the edge client's offset for that topic, invalidating future requests to *get_messages*.

5.4.2 Publish

As depicted in 3, publishing a message for a specific topic results in the edge node storing the message in its topic's corresponding message queue, dispatching the message to the backend node, which then broadcasts the message to all subscribed edge nodes.

5.4.3 Get Messages

To retrieve messages for a topic, clients can leverage **get_messages**, which retrieves all messages for a particular topic from the edge node that the client is attached to. To ensure the edge node does not send duplicate messages to the client upon multiple get_messages calls, that edge node is responsible for keeping track of message offsets, i.e. how many messages that client is yet to consume for a topic. The edge node stores offsets for each edge client, and updates it every time get_messages is called.

5.4.4 Migration

The **migration** API call is used when a mobile edge client identifies they are physically closer to a different edge node, to which they are not yet connected. We assume that mobile clients can use existing techniques to identify the nearest edge node, thus the client can populate the IP address field in the migration request, and has total control over which edge node they wish to be attached to next (or whether they wish to stay attached to the current edge node). As we covered in 5.2.2, migration request will serialize all state updates for a client for every topic they are subscribed to, and leverage ZeroMQ's *request* socket to atomically send the state to a new edge node. The new edge node subscribes to all topics on behalf of the migrated client, instate all state updates in the appropriate message queues, and assigns the client an offset for their un-pulled messages for each topic. Upon successful completion, the new node acknowledges the successful migration, and notifies the old edge node about their ability to locally unsubscribe from all topics on behalf of the migrated client.

6 Evaluation

6.1 Quantitative Evaluations

To test Vertex in a real-world environment, we used Amazon Elastic Kubernetes Service (EKS). In our setup, we provisioned two clusters in the same region (us-east-2), one for a Vertex edge node and another for a Vertex backend node. Both clusters were statically provisioned with m5.large instances each, and the control plane for both clusters consisted of OpenFaas and our shared data and communication pods abstracted as services. We emulated edge clients by running a Python script with our API imported, running at the edge cluster.

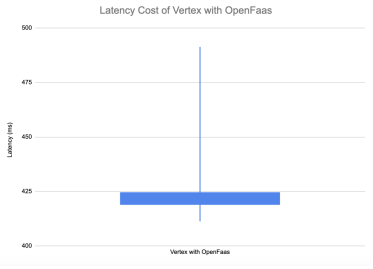
6.1.1 Shared Data and Execution Environment

We first wanted to test the overhead of our serverless engine and shared data subsystem. We initially tested the usage of OpenFaas without alterations, though we expected the performance overhead to be nontrivial. For measurement purposes, we deployed a toy function on the edge node that leverages Vertex's fine-grained shared data abstraction, and uses its linearizability consistency model to increment a shared object (counter). We repeat execution of this function 100 times for our measurement sample size. In our setup, the edge node already has access to the shared object, thus it can execute the function locally when the client triggers the function without coordinating with other edge nodes or the backend node. Despite local processing at the edge node, we noticed large latency overheads from OpenFaas, shown in our boxplot (4b). To supplement the plot with aggregate statistics, we measured the median, mean, and standard deviation of the latency across 100 trials to be 424,28, 427.40, and 14.33 ms, respectively. While the results across invocations were relatively stable, the response latencies for edge applications were overall too high. This prompted us to strip OpenFaas to its barebones and remove any unessential functionalities from our execution environment.

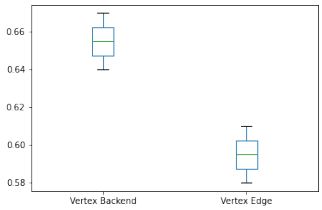
Comparatively, we also measured the network latency from our edge node cluster to our backend node cluster, and the round-trip-time averaged to be 13.1 ms across 100 trials. Basing our empirical findings on prior latency measurements by

	Lines of Code
Object Detection	90
Federated Learning	116

(a) Vertex usability study



(b) Operational latency with unmodified OpenFaas



(c) Shared Data latency

Figure 4: Shared Data and Execution Environment Evaluations

the equivalent Microsoft Azure's service [35], we find that 13ms is reasonable, though on the high end of same-region network latency. Both instances are not in our physical control, but we speculate that they simulate a real-life scenario.

Building off these two measurements, we stripped down OpenFaas and dissected the service to modular pieces, evaluating the necessity of each component to Vertex's correct and agile operation. We ran several modular micro-benchmarks on OpenFaas' components, and found that the largest source for incurred latency was importing and interpreting libraries and packages with each invocation of OpenFaas. Results are shown in 5. We then decided that when running vertex on the edge, all function libraries should be pre-imported and *warmed-up* before execution, to remove the import overheads. We further concluded that OpenFaas' default gateway added significant latency to functions' basic operations, so we extracted the gateway away from the OpenFaas pod (previously dicussed in 3.2).

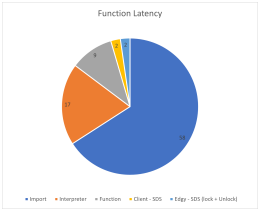


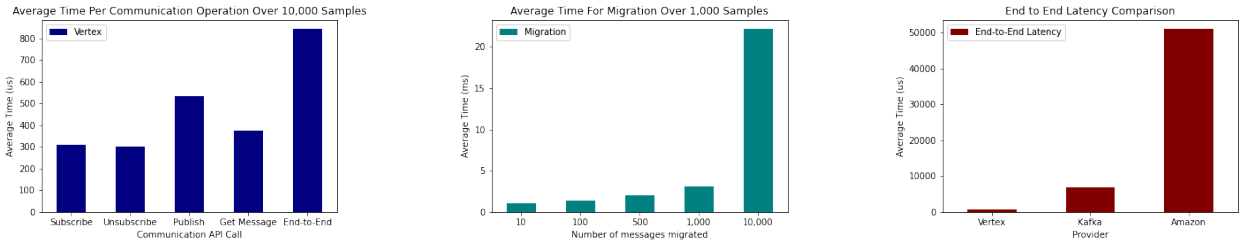
Figure 5: OpenFaas' latency distribution

To fully understand sources of latency within our shared data logic, we performed a simplified evaluation that prevents the underlying serverlesss framework from posing any overhead. Instead of invoking OpenFaas on the clusters to invoke our toy function, we directly invoked the function in Python on the EC2 instances in our clusters. We conducted the experiment first on the edge node, and the remotely on the backend node. Both experiments focused on measuring the shared data abstraction stripped of the execution environment, thus OpenFaas was not used. We repeated the experiment 100 times, and the resulting boxplots are shown in figure 4c. The mean, median, and standard deviation were 65.84, 67,63, and 8.15 ms for the backend, and 60.98, 61.27, 3.66 ms for the edge. We thus concluded that removing warm-up latency from OpenFaas and directing traffic to avoid OpenFaas' gateway should be adopted in our OpenFaas instance.

6.1.2 Communication

Our communication component has a rich set of functionalities that are constructed to satisfy the edge model. We decided to measure pure operation latency, stripped of network latency and propagation delays. This way, we could reason about our subsystem's structure and design rather than on factors out of our reach. Client-edge node interactions were made using gRPC to issue request from the client to the node and responses from the node to the client. We used a commodity AWS EC2 instance (us-east-2) for both the client and the edge node (as separate processes) and opened separate ports for the client, edge node and backend node.

The first measurement was a comprehensive micro-benchmarks for each API supported operation. Each benchmark was repeated 10,000 times, for which we gathered the average, minimum, maximum and 75 percentile times, in microseconds. As witnessed in 6a, **subscribe** operation, from the moment client requested subscription until the edge node sent them acknowledgement of subscription, took 310.97 microseconds on average, with 278.95, 2246 and 314.71 microseconds as the minimum, maximum and 75 percentile, respectively. **Unsubscribe** operation, from client's request to receiving acknowledgement, took 303.03 us on average, and 279.9, 465, and 305.89 as the minimum, maximum and 75 percentile, respectively. **Publish** refers to time between client's request to publish a message and the edge node's acknowledgement after queuing the message in the message queue. Our benchmarks time an average of 532.66 us, with 448.94, 1509 and 563.62 microseconds as the minimum. maximum and 75 percentile latency, respectively. **Get_message** benchmark measured the time between an edge client request to get 100 messages, and receiving the deserialized message stream. All messages were checked for order and correctness. Our measurements revealed 374.68 us on average, and 352.38,



(a) Communication micro-benchmarks (b) Migration latency (c) End-to-End comparison

Figure 6: Communication Evaluation

459, and 378.13 as minimum, maximum and 75 percentiles.

Our next benchmark was the **migration** operation in figure 6b. We created a number of edge clients, populated topic queues with a varying number of messages, and then migrated a client from one edge to another. We repeated this 1,000 times. As a reminder, these measurements do not include network delays, so they concern to the message aggregation per migrating client, serialization, delivery, deserialization, and placement of all messages in the corresponding topic queue on the new edge node. We conducted this measurement with queues containing 10, 100, 500, 1000, 10000 messages. Our average results were 1.02ms, 1.38ms, 2.02ms, 3.08ms and 22.2ms respectively, which is satisfying our edge requirements. We believe that message queues will not contain much larger number of messages due to the ephemeral nature of the communication component.

Our last evaluation for the communication subsystem introduced an end-to-end comparison with some existing industry technologies, shown in figure 6c. We define end-to-end as the overall time it takes between an edge client connected to edge node A publishes a single message, and a different edge client connected to edge node B to get the message and read it. Vertex took 844.97 us on average, over a sample size of 10,000 iterations. Next, we measured Kafka's end-to-end latency, on a local machine running Linux, by creating a single topic, publishing a message, and reading the message with a python client. For 100 iteration, Kafka took 6979 us on average. Next we measured Amazon's pub-sub on the same EC2 instance as Vertex. Every 5 seconds, our client logs the publish time, publishes "PING", and a python client on the other end gets the message and logs the receive time. Over 100 runs, we found Amazon's pub-sub end-to-end latency to be significantly higher, at 51000 us (51 ms).

We conclude our communication component evaluation as a major success; while it is yet to prove similar efficiencies for large volumes (Amazon's pub-sub is said to perform comparatively better for large volume of messages), it shows Vertex's capability of dealing with short-living state updates with high efficiency and ultra low latency.

6.2 Qualitative Evaluations

In addition to our quantitative evaluations, we also evaluated the feasibility of writing edge applications over Vertex, and our results are demonstrated in 4a. We first wrote an object detection application as a Vertex function that uses the YOLO object detection system [39]. The function performs object detection on behalf of an edge client using the YOLOv4-tiny model, and took 90 lines of code to build. We also built a federated learning application on Vertex, where an edge node uses PyTorch [47] in collaborating with other edge nodes to train a model and also allowing clients to make inference requests to this model. This application took us 116 lines to build. While the semantics of Vertex's API are still being refined, we see these results as a positive preliminary indicator of the feasibility of building edge applications over Vertex.

7 Conclusion

In this paper we endeavored to identify the future of edge computing given the changing landscape of technologies and the increasing demands of edge applications in terms of location, computational resources and latency. We enumerated the core requirements of edge applications in the near future, and found that no current offering provides a sufficiently generalized, simple and comprehensive development environment for edge application developers, naturally leading to diminished utilization of the edge despite its virtues. We then proposed Vertex, a unified edge computing environment catered for edge applications, independent of underlying systems used by edge providers and usable for all, preventing vendor lock-in and the limits it imposes on the edge environment. Vertex offers location-agnostic communication, migration, shared data, and agile execution environment as core primitives for edge applications, with straightforward APIs providing maximal flexibility and minimal overheads. We hope to induce interest from cloud and edge providers, ultimately leading to a vast adoption of the unified model and simple primitives, or, at the very least, to a paradigm shift in edge computing that puts edge applications and its requirements at the heart of edge discourse, above other interests and considerations.

References

[1] W. B. Arthur, "Competing technologies, increasing returns, and lock-in by historical events," *The economic journal*, vol. 99, no. 394, pp. 116–131, 1989.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *ACM SIGMOD International Conference on Management of Data*, 1995.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[4] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," vol. 8, IEEE Pervasive Computing, 2009, pp. 14–23.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 63–74.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, pp. 50–58, 2010.

[7] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[8] D. A. Wood, M. Hill, D. Sorin, and V. Nagarajan, *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool Publishers, 2011.

[9] N. Farrington and A. Andreyev, "Facebook's data center network architecture," in *2013 Optical Interconnects Conference*, Citeseer, 2013, pp. 49–50.

[10] D. Petcu, "Multi-cloud: Expectations and current approaches," in *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, 2013, pp. 1–6.

[11] C. Meiklejohn and P. V. Roy, "Lasp: A language for distributed, coordination-free programming," in *17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*, 2015, pp. 184–195.

[12] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[13] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Of Things*, vol. 3, pp. 637–646, 2016.

[14] L. Baresi, D. F. Mendonça, and M. Garriga, "Empowering low-latency applications through a serverless edge computing architecture," in *European Conference on Service-Oriented and Cloud Computing*, Springer, 2017, pp. 196–210.

[15] A. Colley, J. Thebault-Spieker, A. Y. Lin, D. Degraen, B. Fischman, J. Häkkilä, K. Kuehl, V. Nisi, N. J. Nunes, N. Wenig, *et al.*, "The geography of pokémon go: Beneficial and problematic effects on places and movement," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 1179–1192.

[16] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017. DOI: 10.1109/MC.2017.9.

[17] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, "Towards efficient edge cloud augmentation for virtual reality MMOGs," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2017.

[18] T. S. Brisimi, R. Chen, T. Mela, A. Olshevsky, I. C. Paschalidis, and W. Shi, "Federated learning of predictive models from federated electronic health records," *International journal of medical informatics*, vol. 112, pp. 59–67, 2018.

[19] S. L. K. C. Gudi, S. Ojha, B. Johnston, J. Clark, and M.-A. Williams, "Fog robotics for efficient, fluent and robust human-robot interaction," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, IEEE, 2018, pp. 1–5.

[20] Y. Harchol, A. Mushtaq, J. McCauley, A. Panda, and S. Shenker, "Cessna: Resilient edge-computing," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*, 2018, pp. 1–6.

[21] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, "Federated learning for mobile keyboard prediction," *arXiv preprint arXiv:1811.03604*, 2018.

[22] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," in *ACM Symposium on Cloud Computing (SOCC)*, 2018, pp. 401–411.

[23] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino, "Scalability and performance evaluation of edge cloud systems for latency constrained applications," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2018, pp. 286–299.

[24] Q. Chen, X. Ma, S. Tang, J. Guo, Q. Yang, and S. Fu, "F-cooper: Feature based cooperative perception for autonomous vehicle edge computing system using 3d point clouds," in *4th ACM/IEEE Symposium on Edge Computing (SEC 2019)*, 2019, pp. 88–100.

[25] W. Y. Lim, N. C. Luong, D. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, pp. 2031–2063, 2020.

[26] L. Liu, B. Wu, and W. Shi, "A comparison of communication mechanisms in vehicular edge computing," in *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[27] M. Park, K. Bhardwaj, and A. Gavrilovska, "Toward lighter containers for the edge," in *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[28] T. Pfandzelter and D. Bermbach, "Tinyfaas: A lightweight faas platform for edge environments," in *2020 IEEE International Conference on Fog Computing (ICFC)*, 2020, pp. 17–24. DOI: 10.1109/ICFC49376.2020.00011.

[29] A. Trivedi, L. Wang, H. Bal, and A. Iosup, "Sharing and caring of data at the edge," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, USENIX Association, Jun. 2020. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/trivedi.

[30] J. Koch and W. Hao, "An empirical study in edge computing using aws," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 2021, pp. 0542–0549. DOI: 10.1109/CCWC51732.2021.9376039.

[31] Amazon, *Amazon s3*, https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/, Accessed 05-5-2021.

[32] ——, *AWS local zones*, https://aws.amazon.com/about-aws/global-infrastructure/localzones/, Accessed 05-5-2021.

[33] ——, *AWS outposts*, https://aws.amazon.com/outposts/, Accessed 12-5-2020.

[34] ——, *Lambda@edge*, https://aws.amazon.com/lambda/edge/, Accessed 05-5-2021.

[35] M. Azure, *Microsoft azure*, https://docs.microsoft.com/en-us/azure/networking/azure-network-latency, Accessed 05-10-2021.

[36] S. Barua, *Flood of data will get generated in autonomous cars*, https://autotechreview.com/features/flood-of-data-will-get-generated-in-autonomous-cars/, Accessed 05-4-2021.

[37] P. Buffers, *Protocol buffers*, https://developers.google.com/protocol-buffers, Accessed 5-10-2021.

[38] Cloudflare, *Cloudfare workers kv*, https://www.cloudflare.com/products/workers-kv/, Accessed 12-5-2020.

[39] Y. R.-T. O. Detection, https://pjreddie.com/darknet/yolo/, Accessed 05-10-2021.

[40] Google, *Google nest*, https://store.google.com/us/category/connected_home?, Accessed 05-5-2021.

[41] gRPC, *Grpc*, https://grpc.io/, Accessed 5-10-2021.

[42] Kubernetes, *Kubernetes*, https://kubernetes.io/, Accessed 12-5-2020.

[43] Microsoft, *Azure iot edge*, https://azure.microsoft.com/en-us/services/iot-edge/, Accessed 05-5-2021.

[44] MobiledgeX, *Mobiledgex edge-cloud*, https://mobiledgex.com/product, Accessed 12-5-2020.

[45] Mutable, *Mutable*, https://mutable.io/cloud, Accessed 05-4-2021.

[46] OpenFaaS, *Openfaas*, https://www.openfaas.com/, Accessed 12-5-2020.

[47] PyTorch, https://pytorch.org/, Accessed 05-10-2021.

[48] K. Varda, *Introducing cloudflare workers: Run javascript service workers at the edge*, https://blog.cloudflare.com/introducing-cloudflare-workers/, Accessed 12-5-2020.

[49] ZeroMQ, *Zeromq*, https://zeromq.org/, Accessed 05-5-2021.