Vertex: Programming the Edge

Brian Kim

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:

[illegible]

Professor Scott Shenker
Research Advisor

12/16/2020

(Date)

* * * * * * *

[illegible]

Professor Sylvia Ratnasamy
Second Reader

12/16/2020

(Date)

Vertex: Programming The Edge

Brian Kim
UC Berkeley

Abstract

By expanding the computational power of the public cloud to include resources closer to clients, edge computing serves as a natural extension of cloud computing. The resulting reductions in latency and network bandwidth have continued attracting developers to deploy applications at the edge. To accommodate the evolving ecosystem of edge applications, however, the underlying system must be rich in functionality, providing primitives for handling the geo-distributed topology and dynamic environment, while simultaneously delivering on client needs. Current offerings have not been holistic along these dimensions, as core abstractions such as shared data and location-agnostic communication have largely been overlooked. In this paper, we articulate the requirements of edge applications that must be addressed to encourage application evolvability. We follow up by proposing our Vertex system, a programming model and underlying execution environment that provides a unified approach for programming the edge.

1 Introduction

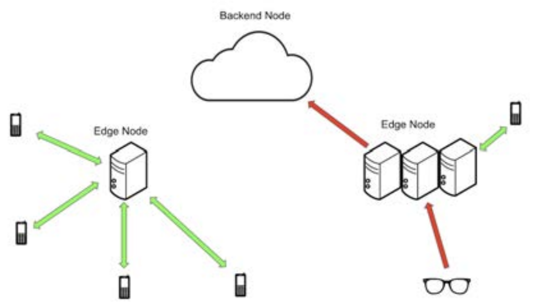
In cloud computing, clients interact with applications deployed in datacenters managed by cloud providers such as Amazon. Edge computing geographically broadens this compute domain, enabling applications to also be deployed on any computational resources closer to clients. The term *edge* is often employed to signify the *network edge*, which broadly describes these physical locations in close proximity to clients, i.e. in reference to where end devices and local networks connect with the public Internet [13]. These resources at the edge are both globally ubiquitous and diverse in nature, ranging from smaller datacenters also managed by cloud providers [4], to server racks in cell towers managed by telecom companies [7] and IoT gateways owned by human consumers [17]. Consequently, edge computing is fundamentally a geo-distributed paradigm, existing over a global topology composed of these distributed compute nodes, and application logic can be partitioned across these nodes.

Though the benefits of utilizing the edge are manifold, the high-level notion of computing closer to clients most intuitively aligns with reductions in latency and network bandwidth. Application logic at the edge can respond to client requests in lieu of cloud datacenters, eradicating the longer latencies incurred by client to cloud communication. In a similar flavor, the amount of traffic leaving the edge can be decreased if the edge application logic can process or filter traffic from clients before the traffic reaches the cloud. These characteristics have continued to entice diverse families of applications that extend beyond edge computing's original roots in serving web content through content delivery networks [16], such as MMO gaming [38], video streaming [22], medical wearables [18], and web security [33].

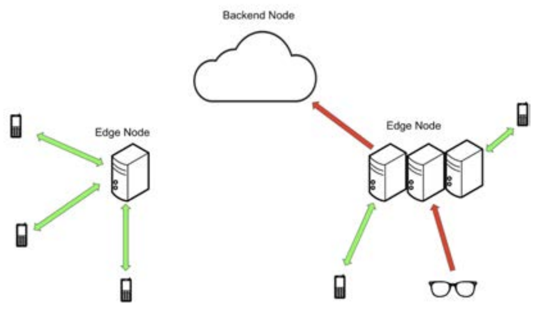
At the same time, however, edge computing is associated with a rich problem space, rendering it difficult to design general-purpose edge systems over which applications can be built and deployed. We proceed to introduce these challenges by roughly grouping them into three general categories. First, there exists environmental challenges specific to the edge and its unique structure as a dynamic extension of the cloud, such as enabling clients to be mobile and migrate from one edge resource to another. Second, as a distributed paradigm, edge computing encounters classic problems in distributed systems, such as consistent access to shared data. Third, edge systems must also meet additional explicit client needs, such as the latency requirements that justify utilizing the edge in the first place.

We claim that the current ecosystem of academic and industrial edge systems does not extensively identify and address these challenges, not only constraining current applications but also stymieing the evolution of future applications. To buttress our claims, we provide discourse on these shortcomings and propose Vertex, a holistic and unified edge system. The remainder of this paper is structured as follows:

- In §2, we motivate Vertex by distilling the edge problem space into more succinct terms (§2.1), articulating the core requirements of edge applications that the underlying sys-



(a) Original topology



(b) Topology after a client moves from one edge node to the other

Figure 1: An example of our edge model, where different colors represent different applications and arrows represent computational flow.

tem must address (§2.2), and demonstrating how the current ecosystem is deficient in these regards (§2.3).

- In §3 and §4, we present the high-level design and implementation of Vertex, respectively, diving into its programming model and underlying execution environment for edge applications.
- In §5, we perform quantitative benchmarks to evaluate Vertex's performance in enabling applications to take advantage of the edge, as well as qualitative evaluations to assess Vertex's adoptability.

2 Motivation

2.1 A Clarified Model of the Edge

Given the eclectic nature of the edge, we endeavor to capture the essence of the problem space by laying out terminology for clarity, and then describing and justifying the assumptions in our model of the edge.

2.1.1 Definitions

Edge clients are end devices that utilize applications deployed at the edge, such as mobile phones or sensors.

Edge nodes are logical units of edge infrastructure over which applications can be deployed, such as network gateways or servers in edge datacenters.

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 organizations that manage and provide the computational resources of *edge nodes* for application developers, such as cloud or network providers.

2.1.2 Assumptions

Certain types of edge nodes are specialized, i.e. they fulfill a single organization's needs or target a certain family of applications. A single company, for instance, may leverage AWS Outposts [5] as a fully managed extension of Amazon cloud services running on the company's on-premise infrastructure. Similarly, IoT gateways [14] often sit near a group of related IoT sensors, aggregating and analyzing the data from these sensors. Instead of these specialized edge nodes, we concern ourselves with multenant edge nodes, i.e. those that are capable of supporting various applications and can be shared among unrelated edge clients, such as servers in CDNs [2,12] or small-scale datacenters for mobile devices [29]. We envision that multitenant edge nodes can help unlock the edge as a publicly accessible and ubiquitous extension of the public cloud, and under this lens, specialized edge nodes represent a degenerate use case. Albeit a separate conversation, multitenancy may also reap other benefits such as economies of scale for edge providers as well as a more streamlined deployment process for applications.

For edge clients, we target those that are mobile, i.e. that are capable of physically moving around at any time, such as medical wearables or mobile devices. Our justifications here are twofold. First, in relation to our targeting of multitenant edge nodes, stationary edge clients (namely fixed IoT sensors and devices) are generally associated with specialized edge nodes that we have excluded from our edge model. Second, stationary clients do not pose any more conceptual difficulties compared to mobile clients, and can be simplified as a degenerate case of the latter.

The notions of multitenancy and mobility broach the additional question of how edge clients are matched to edge nodes. For the degenerate case of stationary clients attached to a specialized edge node, this process is trivial. For the multitenant and mobile case, however, two factors must be addressed when considering this matching mechanism. First, we assume that edge clients being matched to the physically nearest edge node maximizes application utility, i.e. because the primary goal of edge computing is to bring computation closer to clients. Quantitatively proving the soundness of this assumption is orthogonal to our work, and we employ high-level intuition as justification, leaving more rigorous rationale for future work in this area [32]. Second, we assume that existing techniques such as DNS-based mechanisms [31] can be leveraged for clients to learn about nearby edge nodes, as mobility implies that clients may be matched to multiple edge nodes throughout their lifetimes.

Figure 1 demonstrates an example of our edge model. The two colors (green and red) represent two different applications, and the direction of the arrows represent the flow of computation. For example, the green application might represent a mobile application where edge nodes respond to requests from the mobile phones, improving response latency compared to the edge-less case where the phones instead send requests to the cloud backend node. The red 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 backend node for further processing, reducing the amount of network traffic. The difference between subfigures 1a and Figure 1b demonstrates our mobility assumptions, as the mobile phone that moves closer to the other edge node now communicates with that closer edge node.

2.2 Requirements For Edge Applications

While there have been endeavors in academia to identify the needs of edge applications that the underlying system must fulfill [32], clear articulation of these needs is impeded by the diversity of applications that stand to benefit from the edge. For example, applications like MMO gaming [38] are taxing on the distributed nature of the edge, requiring stringent latency needs and coordination across potentially many geo-distributed clients. On the other end of the spectrum, applications like machine learning inference [35] merely leverage the edge as a means of computational offloading, and thus logically represent a single interaction between an edge client and edge node.

For our articulation of application needs, we target applications that fit within the scope of our specific edge model, enabling sharper justification for our proposed needs. Furthermore, we attempt to capture the *core* requirements. As referenced in 1, we utilize three general categories for application needs: requirements stemming from the specific of the structure of the edge (2.2.1), requirements that are more generally observed in distributed systems (2.2.2), and additional requirements that are imposed by edge clients (2.2.3).

2.2.1 Edge-specific Requirements

These application requirements revolve around environmental challenges that are specific to the edge and its structure as a dynamic extension of the cloud.

Migration: As stated in our assumptions (2.1.2), edge clients being matched to the closest edge node maximizes application utility, and we assume existing mechanisms can be employed for clients to learn of nearby edge nodes. Providing applications with the ability to migrate state from one edge node to another gives them control over mobility. For instance, smart cars [9] working together to mark a local map of hazards may desire to migrate to the next edge node if they have traveled substantially far away from the first area. Explicit migration is also necessary for applications where the overhead of immediate migration upon detecting a closer edge node would prove too costly compared to the benefit of attaching to that closer node [25]. An example is AR or VR gaming [38], where immediate migration might disrupt gameplay, and the game application could instead flexibly choose to migrate state across edge nodes while optimizing for user experience.

Location-agnostic communication: In our edge model, clients are mobile and can migrate from edge node to edge node depending on physical proximity. This dynamic environment may also be characterized by abundant edge clients and edge nodes. Furthermore, client and edge node locations for an application are not known *a priori* - clients due to mobility, and edge nodes due to the fact that an application may not need to run on an edge until demand for that application forms nearby. Since the application logic is potentially spread out across edge clients (i.e. the client portions of applications), as well as across 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. An use case that could benefit from location-agnostic communication primitives is social sensing [37] (where data is collected and shared by mobile devices on the behalf of humans, such as disaster reporting applications), as edge clients may desire to communicate to other clients about a particular event.

2.2.2 Distributed System Requirements

These following requirements are associated with classic problems in distributed systems, and generally involve use cases where an application utilizes distributed edge nodes that actually share data among one another.

Consistent access to shared data: For applications where edge nodes operate over a shared dataset, consistent access to data is mandatory. Furthermore, a range of consistency levels must be supplied to match application needs - stronger consistency levels provide more guarantees for shared data access but incur larger overheads to enforce these guarantees, while weaker consistency levels allow data to be left in inconsistent states for the sake of reduced latency. The underlying system must provide expressive primitives across consistency levels for applications to work with shared data. While the need for consistent access to shared data is pervasive across applications, some varied examples include federated learning [21], where weaker consistency may suffice for edge nodes collaborating over a shared machine learning model, and MMO gaming [38], where clients may expect components of the game to remain up-to-date, calling for stronger consistency.

Fault tolerance model: In a similar flavor as data consistency, applications where distributed edge nodes share data must possess a failure model to gracefully recover state if edge nodes go down. In a geo-distributed setting potentially involving many edge nodes, failures will be imminent, and the underlying system must provide applications with well-defined behavior for mitigating failures [23]. Similar to consistent data access, fault tolerance in the face of shared data is a common desire for edge applications. In the context of smart cities [19], for example, where a vast network of edge clients and edge nodes function together to collect and operate on massive amounts of data, one can imagine frequent failures necessitating a set of well-defined fault tolerance protocols.

2.2.3 Explicit Client Requirements

This final category of requirements focuses on additional explicit client needs.

Privacy concerns: Edge computing also opens up a domain in security where clients can specify their privacy needs, e.g. client-driven policies about where data can be placed or accessed among abundant and geo-distributed edge node locations [1]. In particular, the underlying system must provide edge applications with the ability to enforce privacy constraints in the context of other primitives. For example, primitives for shared data may insinuate the movement of data to enforce consistency, but this movement may not always be in compliance with the client's privacy requirements.

Response latency: Finally, the underlying system must be efficient in terms of response latencies, which serves the purpose of leveraging the edge in the first place. In particular, there are two implications here - first, the startup overhead of the execution environment should be fast when clients need to invoke computation at an edge node. Second, the system must ensure that rich semantics to do not come at the expense of response latency. For example, support for mobile clients migrating across edge nodes should impose as little overhead as possible.

2.3 Limitations of Current Offerings

Given these set of core functionalities that edge systems must provide, we claim that the current ecosystem of edge offerings among edge providers is deficient. Especially in industry, these offerings are variegated across edge providers, leading to inconsistent developer experience. A subset of these offerings, i.e. MobiledgeX [24], provide low-level interfaces like VMs and containers to developers. These interfaces provide no primitives for the requirements listed in 2.2, requiring developers to manually deal with these challenges. Other offerings are higher-level in providing function-as-a-service interfaces, such as AWS Lambda@Edge [6] and Cloudflare Workers [11] that may offer core abstractions like shared data [10]. However, there are two primary shortcomings with the primitives that are offered for these abstractions. First, these primitives are often incomplete, e.g. Lambda@Edge leverages s3 blob storage [3] for shared data, which only offers strong read-after-write consistency, or are even non-existent, e.g. migration and communication are sparsely supported. Second, even for the primitives that are provided, the semantics vary across across edge providers, e.g. s3's strong read-after-write eventual consistency versus CloudFlare KV Store's eventual last-writer-wins consistency [10].

Given this ecosystem, we make the observation that edge offerings lack a uniform and extensive approach that holistically deals with migration, communication, consistent and fault-tolerant access to shared data, and privacy. Therefore, current offerings not only fail to generalize to a diverse set of applications, but also stymie the evolvability of future applications. Without a uniform approach, customers are locked into the services of a particular edge provider, as inconsistent semantics across offerings impose a barrier for developers to migrate edge providers - consequently, application developers will be constrained by the semantics and services of a particular offering. Without an extensive approach that provides rich primitives for the edge's needs, writing edge applications be unnecessarily rendered more difficult by the semantics and services of that particular offering. Furthermore, without a uniform and extensive interface, edges clients may not not be able to be matched to the closest edge nodes due to application deployment being restricted to a particular set of edge nodes, which hinders our assumption that clients being matched to the closest edge node maximizes application utility.

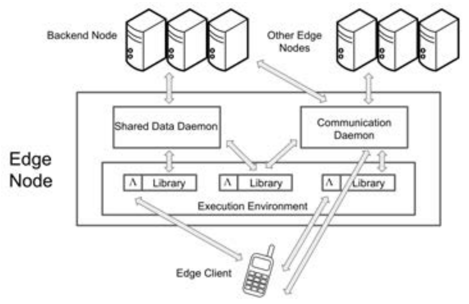
Academic endeavors in edge computing have also been diverse, addressing specific challenges of programming the edge, ranging from Lasp [23], which offers weaker consistency for improved availability and performance in the face of shared data, or TinyFaas [27], which offers a lightweight function-as-a-service platforms for edge nodes for faster execution. Our conversation on academic edge offerings will remain brief and we will focus on industrial offerings, as our claim here is that there exist little endeavors to put all the pieces together for an uniform and extensive edge system that

```
# execution environment
add_function(fn)
run_function(fn_id, args)

# shared data
shared_read(object)
shared_write(object)

# communication
subscribe(topic)
publish(topic, message)
unsubscribe(topic)
get_messages(topic, edge)
migrate(edge)
```

(a) High-level, syntactically sugarized version of Vertex's API.



(b) Vertex system design, where arrows represent computational flow.

Figure 2: Vertex Design

tackles all the core challenges.

To combat these limitations, we present Vertex, an edge system that is a combination of a programming model and an underlying execution environment. The programming model is exposed through a rich API that provides first-class primitives for communication, migration, consistent and fault-tolerant access to shared data, and privacy. The underlying execution environment is necessarily high-level to provide these primitives directly to application developers, and Vertex utilizes a function-as-a-service environment to achieve this interface. We envision that Vertex can be adopted across all edge providers and help unlock the potential of the edge for diverse applications. A detailed design of Vertex is presented in the following section.

3 Vertex Design

Vertex is an edge system that runs across edge clients, edge nodes, and backed nodes. Figure 2a shows the high-level API (implemented as RPC calls) that is exposed to application developers, while figure 2b demonstrates the underlying system. We break up our description into three components as shown in 2b: the underlying execution environment 3.1, shared data 3.2, and communication 3.3.

3.1 Execution Environment

The benefits of cloud-based serverless computing, specifically the Function-as-a-service (FaaS) model, can extend to the edge. In the context of Vertex, the attributes of FaaS conducive to our target edge environment are its performance - invocations that spin up a compute environment must be fast - and the efficient hardware utilization for edge providers through statistical multiplexing. While the Vertex prototype currently leverages OpenFaas [26] due its open-source and well-documented nature, Vertex uses a wrapper around the execution environment so that that any serverless engine can be plugged in underneath. In particular, the API shows two high-level functions, **add_function** and **run_function** that applications developers can use to register their functions to Vertex and allow edge clients to trigger these functions at edge nodes, respectively. Currently, Vertex's prototype supports functions written in Python, and the access to the remaining Vertex subsystems are packaged in as imperative-style libraries with the functions, as demonstrated in 2b. Furthermore, the communication subsystem can be accessed by edge clients directly, as will be demonstrated later in our expanded discussion of communication.

3.2 Shared Data

At the heart of Vertex's contributions is its shared data model, which revolves around strong consistency. 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 strongest consistency from their features. For extensiveness and performance, we are also working to expand Vertex's semantics to weaker levels of consistency. The shared data subsystem is composed of an object-based, **shared_read** and **shared_write** library calls exposed to functions, a daemon running at each edge node that supports Vertex, and additional functionality running on backend nodes that the edge nodes are logically connected to. Application developers can annotate functions with pragmas to signify their consistency needs to Vertex.

3.2.1 Linearizability

To provide linearizability in the face of object-level replication across edge nodes, Vertex leverages the MESI cache coherence protocol [34] and coordinates individual object access across edge nodes. In order to enforce linearizaiblity, Vertex assumes that edge nodes within a general area are attached to one or more backend node that provides this coordination. For performance optimizations in the edge setting, Vertex uses a directory-based write-invalidation protocol that avoids the overhead of broadcasting shared object updates across edge nodes, and instead requires all other replicas of that shared object to be invalidated upon a write. This invalidation approach implies that Vertex maintains a shared object metadata directory which is maintained at the backend node. This directory is maintained at two granularities - first, the backend node maintains the global directory which tracks the coherence states shared objects replicated across edge nodes, and each edge node also maintains a local directory that keeps track of the coherence states of objects at that node.

Vertex achieves linearizability by using compile-time program analysis. By analyzing function pragmas that call for linearizability when an application registers that function using **add_function**, Vertex records a function's read and write set of shared objects and registers this set to the backend node. Our use of the term "function" is overloaded - while we have previously referred to function in the context of FaaS, our prototype currently prescribes modular functions for application developers. For example, a function may be logically modularized into multiple functions, and for each modular function marked with a linearizability pragma, Vertex registers the read and write set of shared objects that the function operates on. We do not prescribe a particular scheme for modularizing functions. For example, each sub-function may be deployed as its own function, or a monolithic function can be deployed that takes arguments to invoke a specific sub-function within it. Moving forward we will refer to these smaller sub-functions as "functions."

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 (i.e. the shared data daemon running on 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 daemon communicates to the backend node that it is missing access to its read and write set, as well as the specific function it is trying to execute.

When the backend node receives this request, the backend checks if it has access to 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. The backend must also check the read set of the requesting edge node, and downgrade replicas to the shared state at other edge nodes that have exclusive access to these objects. Note that these previous steps would be skipped if the backend already has read and write access to these 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's shared data daemon. At that point, barring any other reads and writes from other edge nodes to those objects, further invocations of the function to the original edge node will be executed at the edge node instead of having to go to the backend node.

While potential race conditions across edge nodes are enforced by the MESI invariants maintained by the backend node, Vertex must handle potential race conditions that arise from multiple edge clients concurrently invoking functions *within* the same edge node that may operate on the same shared objects. Currently, Vertex handles these race conditions by enforcing that edge node-local operations related to linearizability (e.g. linearizable function execution, requests to the backend node, or coherence state updates) are executed in a critical section.

Thus far, the conversation on linearizability has focused on consistency guarantees. However, Vertex also ensures that its linearizaiblity mechanism is fault tolerant, another key requirement for edge applications. To achieve this fault tolerance, Vertex leverages leases and barriers, and also assumes that the backend node is failure-free. To deal with failed edge nodes or network partitions, Vertex leverages leases by having edge nodes send heartbeat messages to the backend node. The backend node will invalidate an edge node's access to its shared objects if it does not hear from the edge node within a certain time period, and gives itself the coherence states over those objects instead, i.e. if the failed edge node had exclusive access to an object, the backend node now has exclusive access to that object. Conversely, edge nodes can also detect failures (i.e. network partitions), since they expect the backend to send backs acknowledgements upon receiving heartbeat messages. If no acknowledgements are received within a certain time period, the edge node will no longer service edge client requests until the network partition is resolved.

However, with only leases, Vertex cannot recover shared object state that is lost if an edge goes down. To rectify this, Vertex supplements leases with barriers. More specifically, functions that write to shared objects are forced to write back to the backend node, and nothing is returned to the edge client that invoked the function until this value has been written back and acknowledged by the backend node. This use of barriers guarantees that any function invocation will have necessarily been successful in persisting updates to data.

3.2.2 Serializability

Vertex's linearizability feature is apt for short-lived operations on shared objects, and 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. However, for functions consisting of potentially many reads and writes, the application developer may find serializability to be a more fitting mechanism for enforcing strong consistency. At a high-level, Vertex allows functions to be marked as needing serializabilty, and instead of bring *data to compute*, Vertex brings *compute to data*, allowing functions to lock a set of objects and perform computations locally at edge nodes when triggered by edge clients. Vertex also leverages caching of objects to improve performance.

In particular, Vertex's serializability mechanism enforces snapshot isolation, [8] which provides most of of the consistency guarantees of serializabilty while also improving performance. In order to achieve snapshot isolation, Vertex assigns globally unique logical start times to functions invocations marked with the serializability pragma. Throughout its invocation, 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 - first, the function is the only one writing to the object, and second, 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. Furthermore, Vertex tracks the start time of the function who last read an object so that functions that need write access can be guaranteed to have a logical start time later than the logical start time of this last read. If this condition is not met when trying to acquire the lock, the function is aborted and restarted. Finally, on top of these guarantees, Vertex ensures that writes are atomic, i.e. either all or none of a function's writes become available to other functions.

To more generally comment on these consistency guarantees, the combination of Vertex's atomicity and ensuring that objects read by a function have been written after the read started provides snapshot isolation. Compared to serializability, which requires analyzing the of reads and writes of functions when determining whether to abort or to commit, snapshot isolation only requires the write set to be tracked, leading to improved performance. Vertex further improves performance by caching objects at edge nodes, ameliorating the overhead incurred by accessing the underlying object store. These caches are lazily sychronized to the backend node. To achieve all these functionalities, Vertex utilizes a backend node to run a metadata coordinator that controls and tracks information like logical start times for functions across edge nodes. Daemons running at edge nodes communicate with the backend node to determine object accesses, enabling functions to commit/abort, and providing Vertex's caching feature.

In addition to consistency guarantees, Vertex ensures that snapshot isolation is provided in a fault-tolerant manner. The metadata coordinator running on backend nodes is replicated, and the primary coordinator ensures that it backs up its state at the secondary coordinator. When daemons running on edge nodes cannot connect to the primary coordinator, they communicate with the secondary coordinator, which makes sure that the primary has failed and services the edge nodes daemons in lieu of the primary. In addition to metadata coordinator failures, Vertex also deals with edge node daemon failures, which may lead to a loss of state stemming from daemon's role of handling object access through locks and enabling functions to commit and abort (e.g. a downed daemon can no longer release shared object locks for functions). Vertex utilizes leases and timeouts to deal with these potential problems. Instead of vanilla locks, the daemons use a leased version of locks instead so that if the daemon times out, the locks are automatically released. Furthermore, if functions have not successfully returned due to being blocked by a downed daemon, Vertex uses the timeout to restart these functions. Finally, to deal with potential data loss that results if a daemon goes down (i.e. if data cached at an edge node that had yet to be persisted is lost), Vertex leverages lineage-based reconstruction - multiple versions of an object are stored, and upon a daemon failure, Vertex can reconstruct the lost data by observing lineage information. Lineage is encoded as a list of objects accessed by a function as well as timestamps for when the function modified the data, and when the daemon comes back up, it reexecutes functions in the lineage-based order that enables the recovery of the lost data.

3.2.3 Weaker Consistency Models

To provide an expressive range of consistency guarantees and improved performance, we are working on integrating weaker forms of consistency into Vertex's shared data subsystem. Currently, Vertex's baseline model of eventual consistency 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 synchronizes potentially stale versions of shard objects using the merge functions across versions across edge nodes. In order to provide a more rigorous form of eventual consistency, we have also implemented different CRDTs (i.e. a grow-counter) using the underlying locking and broadcast primitives that are used to implement Vertex's linearizability. However, we are still considering how we would incorporate CRDTs [30] into Vertex's API with minimal interference. Finally, we have found that eventual consistency, causal consistency, and strong consistency are generally sufficient to meet the diverse needs of edge application, and are planning on incorporating causal consistency into Vertex's shared data module as well.

3.3 Communication

While shared data implies some persistent state, applications also may require more ephemeral primitives for communication across edge clients and edge nodes. In particular, communication must be location-agnostic, as outlined in 2.2.1, and account for edge client mobility, as edge clients that attaches to another edge node due to physical proximity must not interrupt the semantics of communication. To achieve this, part of Vertex's API includes a pub-sub interface, accompanied by a primitive for migrating communication state. Vertex's communication subsystem leverages ZeroMQ [36] as the underlying messaging library but leverages a wrapper over it so any underlying messaging library can be utilized.

3.3.1 Pub-Sub Communication

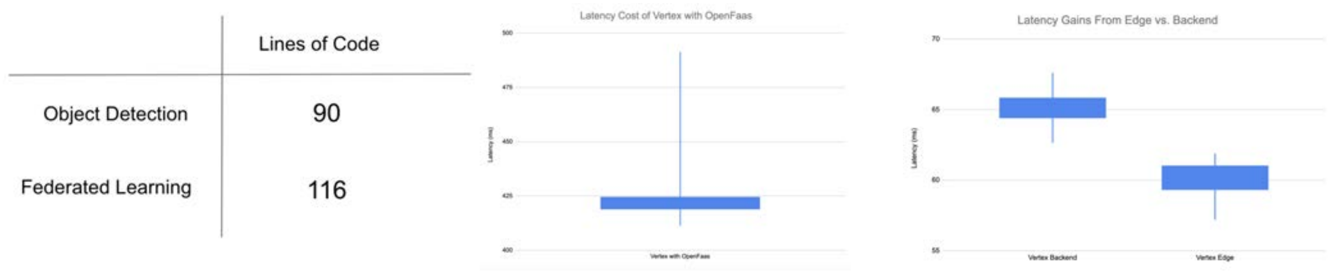
As demonstrated in 2a, developers have a familiar set of pub-sub tools to build their applications. **subscribe** allows any endpoint, i.e. an edge client or an edge node, to subscribe to a named topic, while **publish** allows an endpoint subscribed to a particular topic to publish a message to all endpoints subscribed to that topic. The nuance of Vertex'a pub-sub model is how messages are stored and broadcast to subscribed endpoints, a mechanism that makes use of an additional backend node. When an edge client subscribes to a topic, the edge node it is currently matched to (which is the closest edge node, as stated in our assumptions) implicitly also subscribes to that topic if it has already not done so. That edge node then checks with the backed node to see if that topic has already been registered, and if not, registers that topic with that backend so that other edges can subscribe to the same topic. When a client or edge node publishes to a topic, the edge node records the message, associating the message with the topic, and also persists the message to the backend, which is responsible for publishing the message to all subscribed edge nodes.

In effect, the edge node acts as a message aggrregator on behalf of clients, while the backend node serves as the aggregator for edge nodes. Note that while we have described how edge nodes receive messages for subscribed topics from the backend node, we have thus far has omitted the details of how edge clients can receive messages for topics they are subscribed to. 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 (which we assume is the closest edge node). 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. for each client attached to that edge node, the edge node maintains message offset information (where the offset represents how many messages that client has consumed for a topic) for all topics that client is subscribed to. Finally, **unsubscribe** allows either edge clients or nodes to unsubscribe from a particular topic. Another topic of discussion is how **subscribe**, **publish**, and **unsubscribe** do not require another parameter for clients to specify which edge to perform these actions on - this is due to our assumption that edge clients are matched to the nearest edge node, thus these calls are routed to that nearest edge node. In a similar flavor, client calls to **get_messages** default to the nearest edge node as well; however, we allow endpoints (clients or edge nodes) to specify any specific edge to retrieve messages from. This makes our pub-sub mechanism flexible by enabling any endpoint to query an edge node within the system for the messages associated with a particular topic. Furthermore, Vertex also utilizes timeouts, i.e. if a particular endpoint does not check in with the communication subsystem, they are implicitly unsubscribed from their topics. Once all subscribers have read all updates to a particular topic, or all subscribes have timed out, then the edge node can proceed to prune the messages from its internal state.

We observe that this mechanism achieves location-agnostic communication in two granularities. From the edge client perspective, they do not know the locations of other edge clients *a priori*, but can leverage Vertex's communication daemon on edge nodes as a means of publishing and subscribing to topics. 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 edge nodes - however, by leveraging the datacenter as a rendezvous point, edge nodes can publish and subscribe to topics across one another, allowing them to implement the functionalities to provide location-agnostic communication for edge clients. As a final note on Vertex's pub-sub, we note that there are two entry points into Vertex's communication subsystem. While we have addressed edge clients directly invoking the edge node communication subsystem, functions deployed and triggered on edge node may also talk to the subsystem, using the Vertex library hooked into functions.

3.3.2 Migration

Our assumptions of client mobility 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 channels it is subscribed



	Lines of Code
Object Detection	90
Federated Learning	116

(a) Vertex usability study (b) Cost of Vertex with OpenFaas (c) Latency Gains From Vertex

Figure 3: Various Vertex Evaluations

to as well as its message offsets are stored at its previous edge node. To address this, the Vertex API supports client-driven mobility enabling clients to migrate their communication state to the new, closer edge node. The combination of Vertex's pub-sub functionality and migration primitive enables communication to be fully decoupled from location. Note that in our assumptions, clients can learn about nearby edge nodes using existing techniques, thus they can specify the new edge address as a parameter to **migrate**. Internally, a migrate call from edge node A to edge node B implies that node A will notify node B about all topics that the client is subscribed to, messages that the client has yet to consume for subscribed topics, as well as message offsets per topic, so node B can take over in servicing the client for communication. If edge B does is not already subscribed to a topic sent over from a migrate call, it will itself subscribe to this topic so it can continue to service the client.

3.4 Privacy

While we have briefly touched on edge client privacy needs in 2.2.3, the clear semantics and functionalities of Vertex's privacy subsystem are still a work in progress. In addition to general privacy policies, we are also specifically considering how client privacy needs may affect Vertex's shared data and communication subsystems, i.e. bringing data to compute for linearizability may violate client privacy needs.

4 Implementation

Being a holistic edge framework that applications can be deployed over, Vertex runs across edge clients, edge nodes, and backend nodes. Edge nodes that support Vertex are organized as Kubernetes [20] clusters, each of which runs an instance of a serverless engine (which is currently OpenFaas). For the remaining Vertex subsystems, i.e. the shared data and communication daemons on edge nodes, we use a microservices-based approach, where each of these subsystems are deployed on a dedicated Kubernetes pod. Furthermore, the components of Vertex that run on backend nodes also assumes that backend nodes are organized as Kubernetes clusters that run a serverless engine. The reason for this assumption stems from the fact that Vertex's linearizability feature implies that the backend node may execute functions if the edge node does not have access to a shared object. Similar to Vertex edge nodes, backend edge nodes running as clusters also have communication and shared data components running on dedicated pods.

All communication between Vertex components is achieved using RPC calls. Both edge clients (which are currently emulated as Python client programs) as well as functions deployed at edge nodes can invoke Vertex's API as RPC calls. Since we leverage OpenFaas as our underlying serverless framework, we package in RPC libraries along with function code so they can invoke the API. The pub-sub communication module, built around ZeroMQ, is implemented in C++ and wrapped in a Python RPC library and exports the API calls described in 3.3. The execution environment wraps OpenFaas with a Python RPC library, exporting the API calls described in 3.1. For the shared data module, linearizability and eventual consistency are implemented in Rust, while serializability is implemented in C++, and these are also wrapped in a Python RPC library that exports the API calls described in 3.2. Furthermore, Vertex edge nodes can talk to the backend edge nodes using RPC calls to implement the edge to backend mechanisms (e.g. cachce coherence traffic for linearizability).

5 Evaluation

5.1 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 these 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. To emulate a client, we wrote a client script on the EKS cluster representing the edge node.

We first wanted to test the overhead of our serverless engine - while we used OpenFaas for its convenient usage, we expected the performance overhead to be nontrivial. To measure this, we deployed a toy function on the edge that leverages Vertex's linearizability feature and increments a shared object (a counter). The client function on the edge triggers the function, and we repeat this 100 times for our measurement. 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. Despite local processing at the edge node, we noticed huge latency overheads, which we observed to be from OpenFaas, shown in our boxplot (3b). 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 proves our conjecture that the choice and design of FaaS frameworks on the edge require require more thought.

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. We note two caveats about this measurements. First, the locations of the clusters within the same AWS region were not within our control, and second, more investigation is needed to if these network latency numbers are realistic in a real world scenario where an edge node communicates with a backend node in the cloud.

Building off these two measurements, we performed an evaluation that represents an ideal scenario where the underlying serverlesss framework poses no 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 had two setups for this experiment. First, the client, which is attached to the edge, invokes the toy function on the edge cluster, which increments the shared object counter directly without going through OpenFaas. Second, the client instead invokes the function across the network on the backend node instead, which also increments the shared object counter without using OpenFaas.

The goal of this experiment was to observe the latency gains from directly running the functions, as well as to compare the scenarios when a client does not use the edge node versus when the client uses the cloud instead for the same computation. For both the edge and the backend, we repeated the experiment 100 times, and the resulting boxplots are shown in figure 3c. 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. While further performance optimizations and fine-tuning of our setup will are necessary, these results demonstrate the potential that computing on the edge offers over computing at the cloud.

Going forward, the immediate next steps in our evaluation process are to deploy another edge node cluster on EKS, so we can more expressively evaluate our shared data and communication subsystems, which are particularly useful when there are multiple, collaborating edge nodes. Furthermore, while integrating Vertex with EKS, we found a range of performance bugs that did not manifest themselves in a local setup, especially in our shared data subsystem, and we plan to continue fixing these bugs.

5.2 Qualitative Evaluations

In addition to our quantitative evaluations, we also wanted to evaluate the feasability of writing edge applications over Vertex, and our results are demonstrated in 3a. We first wrote an object detection application as a Vertex function that uses the YOLO object detection system [15]. 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 as a Vertex application, where an edge node uses PyTorch [28] 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.

6 Conclusion

In this paper we have endeavored to identify location-agnostic communication, migration, shared data, and privacy as core requirements for edge applications which have been overlooked by industrial offerings. We follow up by presenting Vertex, an edge framework that represents a unified and extensive environment for deploying edge applications by providing primitives for these core requirements. While we plan to continue working on improving Vertex's performance and the expressivity of its API, we hope that it can serve as a stepping stone to reimagine how a unified edge environment that supports a rich range of applications can unlock the potential benefits of the edge.

References

[1] A. Alwarafy, K. A. Al-Thelaya, M. Abdallah, J. Schneider, and M. Hamdi. A survey on security and privacy issues in edge computing-assisted internet of things. *IEEE Internet of Things Journal (Early Access)*, pages 1–1, 2020.

[2] Amazon. Amazon cloudfront. https://aws.amazon.com/cloudfront/. Accessed 12-5-2020.

[3] Amazon. Amazon s3. https://aws.amazon.com/s3/. Accessed 12-5-2020.

[4] Amazon. AWS local zones. https://aws.amazon.com/about-aws/global-infrastructure/localzones/. Accessed 12-5-2020.

[5] Amazon. AWS outposts. https://aws.amazon.com/outposts/. Accessed 12-5-2020.

[6] Amazon. Lambda@edge. https://aws.amazon.com/lambda/edge/. Accessed 12-5-2020.

[7] AT&T. AT&T multi-access edge computing. https://www.business.att.com/products/multi-access-edge-computing.html. Accessed 12-5-2020.

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD International Conference on Management of Data*, 1995.

[9] Qi Chen, Xu Ma, Sihai Tang, Jingda Guo, Qing Yang, and Song 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)*, pages 88–100, 2019.

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

[11] Cloudflare. Cloudflare workers. https://workers.cloudflare.com/. Accessed 12-5-2020.

[12] Cloudflare. What is a data center? https://www.cloudflare.com/learning/cdn/glossary/data-center/. Accessed 12-5-2020.

[13] Cloudflare. What is edge computing? https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/. Accessed 12-5-2020.

[14] Dell. Dell edge gateways for iot. https://www.dell.com/en-us/work/shop/gateways-embedded-computing/sf/edge-gateway/. Accessed 12-5-2020.

[15] YOLO: Real-Time Object Detection. https://pjreddie.com/darknet/yolo/. Accessed 12-5-2020.

[16] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6:50–58, November 2002.

[17] Google. Google nest. https://store.google.com/us/category/connected_home? Accessed 12-5-2020.

[18] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. IONN: Incremental offloading of neural network computations from mobile devices to edge servers. In *ACM Symposium on Cloud Computing (SOCC)*, pages 401–411, 2018.

[19] L. U. Khan, I. Yaqoob, N. H. Tran, S. M. A. Kazmi, T. N. Dang, and C. S. Hong. Edge-computing-enabled smart cities: A comprehensive survey. *IEEE Internet of Things Journal*, 7(10):10200–10232, 2020.

[20] Kubernetes. Kubernetes. https://kubernetes.io/. Accessed 12-5-2020.

[21] Wei Yang Lim, Nguyen Cong Luong, D. Hoang, Y. Jiao, Ying-Chang Liang, Qiang Yang, D. Niyato, and Chunyan Miao. Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22:2031–2063, 2020.

[22] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *ACM/IEEE Symposium on Edge Computing (SEC)*, pages 286–299, 2018.

[23] Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, coordination-free programming. In *17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*, pages 184–195, 2015.

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

[25] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadil. DPaxos: Managing data closer to users for low-latency and mobile applications. In *ACM International Conference on Management of Data (SIGCOMM)*, pages 1221–1236, 2018.

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

[27] T. Pfandzelter and D. Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24, 2020.

[28] PyTorch. https://pytorch.org/. Accessed 12-5-2020.

[29] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. volume 8, pages 14–23. IEEE Pervasive Computing, 2009.

[30] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. In *SSS*, 2011.

[31] Masaki Suzuki, Takuya Miyasaka, Debashish Purkayastha, Yonggang Fang, Qiang Huang, Jinguo Zhu, Balendu Burla, Xiaopeng Tong, Dan Druta, Jane Shen, Hanyu Ding, Guo Song, Marco Angaroni, and Viscardo Costa. Enhanced DNS support towards distributed MEC environment, 9 2020. Whitepaper No. 29.

[32] Animesh Trivedi, Lin Wang, Henri Bal, and Alexandru Iosup. Sharing and caring of data at the edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

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

[34] David A. Wood, Mark Hill, Daniel Sorin, and Vijay Nagarajan. *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool Publishers, 2011.

[35] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[36] ZeroMQ. Zeromq. https://zeromq.org/. Accessed 12-5-2020.

[37] D. Zhang, N. Vance, and D. Wang. When social sensing meets edge computing: Vision and challenges. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2019.

[38] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. Towards efficient edge cloud augmentation for virtual reality MMOGs. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2017.