BumbleBee: Application-aware adaptation for edge-cloud orchestration

HyunJong Lee, University of Michigan
Shadi Noghabi, Microsoft Research
Brian Noble, University of Michigan
Matthew Furlong, Microsoft
Landon P. Cox, Microsoft

***Abstract*—Modern developers rely on container-orchestration frameworks like Kubernetes to deploy and manage hybrid workloads that span the edge and cloud. When network conditions between the edge and cloud change unexpectedly, a workload must *adapt* its internal behavior. Unfortunately, container-orchestration frameworks do not offer an easy way to express, deploy, and manage adaptation strategies. As a result, fine-tuning or modifying a workload's adaptive behavior can require modifying containers built from large, complex codebases that may be maintained by separate development teams. This paper presents BumbleBee, a lightweight extension for container-orchestration frameworks that separates the concerns of application logic and adaptation logic. BumbleBee provides a simple in-network programming abstraction for making decisions about network data using application semantics. Experiments with a BumbleBee prototype show that edge ML-workloads can adapt to network variability and survive disconnections, edge stream-processing workloads can improve benchmark results between 37.8% and 23x, and HLS video-streaming can reduce stalled playback by 77%.**

I. Introduction

Hybrid workloads that span edges and clouds are on the rise [24], [49]. Container technologies like Docker [51] and orchestration platforms like Kubernetes [31] are crucial to hybrid workloads because they provide a uniform compute and control plane. Orchestrators can launch tasks to satisfy bursts of new requests, kill tasks when utilization drops, and load-balance traffic. However, even perfect orchestration cannot ensure good performance and reliability in the face of highly variable network conditions.

This limitation is acute for hybrid workloads because, unlike within a single cluster, network conditions between the edge and cloud can change unexpectedly [13], [55], [77] and partitions are not uncommon [1], [4], [6], [26], [27], [52]. When network conditions degrade it is crucial for workloads to adapt their internal behavior in response. For example, a machine-learning (ML) workload may switch to an edge inference model with lower accuracy to compensate for higher network latency, and a stream-processing workload may aggregate more aggressively to compensate for a drop in network bandwidth. In these cases and many others, *application-aware adaptation* [54] is the key to maintaining acceptable quality when network conditions degrade.

Unfortunately, applying application-aware adaptation to orchestrated workloads is onerous. At the orchestration level, tools like Azure Arc [5], Google Anthos [25], AWS Hybrid Cloud [3], and KubeEdge [40] provide a centralized control plane for hybrid workloads, but they do not support adaptation. At the application level, adaptation strategies are often tightly coupled with other functionality in a single container, such as a video-processing container that implements adaptive bitrate logic and video transcoding. As a result, fine-tuning or modifying a workload's adaptive behavior can require changes to a large codebase that is often maintained by a separate development team. At the network-transport level application-oblivious responses to variable network conditions, such as TCP congestion control, provide fair bandwidth allocation, but only the application knows how to change its internal behavior as conditions change.

To fill this gap, we present a lightweight in-network processing facility for application-aware adaptation called *BumbleBee*. BumbleBee provides a clean *separation of concerns* between workloads' adaptation and business logic. Workloads' core functionality remain in their original unmodified containers, and BumbleBee adaptation scripts execute in sidecar proxies. BumbleBee benefits a variety of hybrid workloads: ML applications can gracefully switch between high- and low-fidelity inference, stream-processing applications can meet between 37.8% and 23x more deadlines, and video-streaming applications can reduce stalling by 77%.

The main technical challenge that BumbleBee addresses is balancing expressiveness and modularity. Embedding adaptation within an application container allows arbitrary expressiveness but provides poor modularity since business and adaptation logic are trapped in the same component. At the other end of the spectrum, P4 [9] can be applied to unmodified workloads but provides a very limited programming model. Envoy sidecar [74] filters are more expressive than P4, but they cannot perform stateful adaptations such as redirecting or transforming messages based on how long they have been delayed.

BumbleBee allows developers to write concise scripts, often only tens of lines of code, that can programmatically drop, redirect, reorder, and transform network data. Heavyweight computations, such as video transcoding, can be asynchronously offloaded, and BumbleBee handles reintegrating these modified payloads into a workload's message queue. Because scripts execute within a user-level sidecar proxy, BumbleBee can adapt workloads without modifying existing containers.

This small incremental change to an existing mechanism allows application-aware adaptation to benefit from the separation of concerns already provided by the container ecosystem. Applications need not take on the task of network monitoring, and adaptation strategies can be updated and repurposed without inspecting complex source code or deploying new containers. BumbleBee's approach also allows placement of adaptive mechanisms closer to the point at which change occurs in the network, improving agility over more end-to-end approaches.

2022 IEEE/ACM 7th Symposium on Edge Computing (SEC) | 978-1-6654-8611-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SEC54971.2022.00017

978-1-6654-8611-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SEC54971.2022.00017

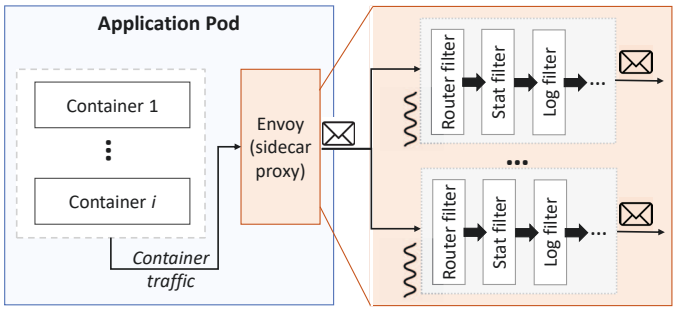


Fig. 1: Envoy sidecars interpose on a pod's network communication.

This paper makes the following contributions:

- We identify four common patterns applications use to adapt to network variability.
- We design and implement a single in-network abstraction to implement all of these patterns, informed by application needs.
- Experiments with our prototype show that edge workloads benefit from BumbleBee: (1) ML applications can utilize cloud resources when available and operate without interruption when disconnected, (2) BumbleBee increases the number of deadlines met between 37.8% and 23x on the Yahoo! stream-processing benchmark, (3) BumbleBee reduces stalled playback by 77% during HLS video streaming under real-world network conditions, and (4) BumbleBee adds less than 10% overhead to the 99th percentile request latency compared to a baseline sidecar.

The rest of the paper is organized as follows. Section II describes background information. Section III describes the BumbleBee design. Section IV presents an evaluation of our BumbleBee prototype. Section V describes related work. And Section VI presents our conclusions.

II. Background

Applications are increasingly written in a *containerized framework* [8], [51], [61] as a collection of communicating *microservices* [53]. These frameworks provide many advantages: a strict decomposition of tasks, a consistent deployment model allowing in-place updates, declarative capture and preservation of system dependencies, and lightweight resource isolation and monitoring. Such ecosystems explicitly provide for *separation of concerns* through architectural decisions. Application writers need not be concerned with task creation, monitoring, placement, or scaling, relying instead on *container orchestration frameworks* [31], [71]. Likewise, they need not actively manage the communication between emplaced tasks. Instead, a *service mesh* [44], [60] provides reliable, fault-tolerant, load-balanced communication across complex topologies of task deployment. This section describes these frameworks, with an eye to BumbleBee's integration with them.

Containers: Docker [51] is a container-based virtualization platform that provides process-level performance and security isolation; such platforms have become the standard unit to manage and deploy software in the cloud. Container images include all of the user-level state required to launch an application, including binaries, support libraries, and configuration. Each container typically implements a single component microservice of the overall application, providing an API to the other constituent components.

Container Orchestration: Kubernetes [31] automates deployment, scaling, and management of distributed, containerized applications. The unit of deployment in Kubernetes is a *pod*. A pod is a set of containers that run under the same operating system kernel and share the same underlying physical resources, such as cores and disks. Because containers within a pod share a machine they can communicate cheaply via local storage or intra-kernel messaging.

Developers write configuration manifests describing how Kubernetes should deploy an application on a set of physical or virtual machines, e.g., which container images to use, how containers are grouped into pods, and which ports each pod needs. The manifest also describes runtime goals for an application, such as pod replication factors, load balancing among replicas, and an auto-scaling policy.

Service Mesh: Service meshes [44] manage inter-pod communications within Kubernetes. They provide service discovery, peer health monitoring, routing, load balancing, authentication, and authorization. This is done via the *sidecar pattern* [10], in which a user-level network proxy called Envoy [39] is transparently interposed between each pod and its connection to the rest of the system; applications are oblivious to the sidecar and its mechanisms. Each Envoy instance is populated with `iptable` rules to route incoming and outgoing packets through the sidecar, as shown in Figure 1. This architecture makes the Envoy sidecar an ideal place to implement application-aware adaptation. It allows application writers to focus only on the needs of adaptation as data traverses the network, without having to integrate it with the application's behavior as prior systems did [23], [54]. We use the Istio [60] implementation in our prototype.

An Envoy sidecar has a pool of worker threads, mapped to the underlying threads exposed to this container. Workers block on ingress/egress sockets, and are invoked on a per-message basis. On invocation, the Envoy worker passes the message through one or more application-specific *filters*. Filters are small, stateless code snippets that operate on individual messages. Filters have full access to a message and can perform simple operations, such as redirection, dropping, and payload transformation. Developers commonly use Envoy filters for monitoring and traffic shaping, such as collecting telemetry, load-balancing, and performing A/B testing. Envoy supports filters at several layers of the network stack.

III. Design and Implementation

In designing BumbleBee, we kept three goals in mind. First, we followed the container ecosystem's core principle of *separation of concerns*, isolating the application's logic from adaptation decisions. The existing infrastructure of orchestration frameworks and service meshes made this particularly

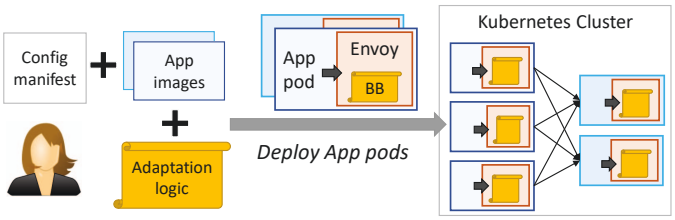


Fig. 2: Kubernetes deployment of a distributed application with BumbleBee enabled.

attractive. Second, we *kept interfaces as narrow as possible*. There were places where BumbleBee needs some additional information or functionality, but those were chosen only reluctantly. Third, we erred on the side of *simple and inexpensive* in designing the interface exposed by BumbleBee whenever possible.

BumbleBee's overall architecture is illustrated in Figure 2. Authors define applications through a YAML manifest that describes the set of application container images and their corresponding configurations. When deployed, the Istio service mesh co-locates an Envoy sidecar proxy with each Kubernetes pod; this sidecar is interposed on all traffic to and from the pod. Applications supply BumbleBee adaptation logic as simple Lua scripts [35], deployed in the sidecars as Envoy filters [60]. Cilium [16] and eBPF [73] provide functionality similar to Envoy filters and could be used for an alternative BumbleBee implementation. All elements in a container ecosystem can be updated in place; thus these scripts can be changed on the fly without stopping or re-deploying the overall application.

To determine how best to frame the abstractions provided by BumbleBee, we surveyed a variety of existing adaptive systems in various domains:

- *Video streaming:* live or on-demand videos are streamed from a server to a client. When bandwidth is constrained, parties involved *transform* streams to lower video resolution, or *drop* frames-per-second [72].
- *Video conferencing:* multiple users interactively video-chat. When network is congested, clients *transform* frames to lower resolutions [19], [79], or *drop* to voice-only mode (e.g., FaceTime "poor connection" [59]). For consistent poor network conditions, they *redirect* to calls through a different meet-up server [37], or occasionally *drop* out-of-sync frames if delayed too long.
- *Internet of Things (IoT):* distributed clients collect and stream sensor data to the cloud. The clients react to changes in network environment by dropping (filtering) data points below a threshold [28], aggregating or *tranforming* data into an average or histogram [36], and *re-ordering* low-priority data points to the tail of the queue [30].
- *Stream-processing applications:* distributed workers process *continuous* streams of data for real-time insights (e.g., fraud detection). However, the distributed nature of the jobs often encounter turbulent network environment. The applications adapt by *dropping* less important messages [64], *re-ordering* new messages to the front queue to avoid backlogs [2], [58], *transforming* multiple messages to aggregated message [66], and *redirecting* messages that are assigned to overloaded worker to another worker.

We identified four common *adaptation patterns* used across existing adaptive systems:

- *Drop* eliminates data when it is no longer useful.
- *Reorder* defers lower priority data in preference to more important but subsequent elements.
- *Redirect* changes routing from an over-utilized resource to an available but possibly lower-quality one.
- *Transform* converts data from one format to another, typically reducing size at the cost of data fidelity.

Each system we examined used at least one of these patterns; some combined more than one. Importantly, all of these can be implemented by observing at most a small, contiguous range of network messages at the head of the current transmission sequence. Stock Envoy exposes messages individually to stateless scripts. We expanded this interface to allow scripts access to a single mutable ordered queue for each (source, destination) pair. We were reluctant to widen the interface in this way, but doing so is necessary to support reordering of messages in applications that can benefit from it.

A. BumbleBee's interface

BumbleBee aims to provide an interface that is both simple and low-overhead. This motivated a few key design decisions. The first was to describe adaptation strategies via an imperative Lua scripting interface. We initially explored declarative interfaces like YAML or SQL. Unfortunately, we found it difficult to express simple adaptations declaratively. At the same time, these systems included significant unnecessary mechanism, representing a potential runtime liability to the critical path of message processing. In addition, nearly all individual adaptation strategies of which we are aware of have been implemented in imperative languages, and it seemed burdensome to change models.

Second, we explicitly do not support the use of Lua libraries beyond the standard, built-in set. This helps ensure that in-script behavior is simple, inexpensive, and reduces the surface area for malicious actors. If more complex functionality is necessary, it must be provided through external callbacks, as we discuss below.

BumbleBee views the world from the perspective of a single Kubernetes pod, and how that pod should react to changes in the network. Figure 3 shows a sample BumbleBee adaptation script for a surface street traffic monitoring application. This application uses an ML model to detect traffic, choosing between an inexpensive one on the nearby edge or a remote, full-fidelity version in the cloud. It adaptively decides where to run the ML model and at what resolution. As shown, the main abstraction exported to BumbleBee scripts is a set of queued messages per (source, destination) pair. The scripts can

```
1  function envoy_on_request(h)
2    -- for each sink
3    for queue in h:Queues():getQueue() do
4      -- check the route
5      route = queue:route()
6      if string.find(route, "cloud") then
7        -- check current bandwidth estimate
8        bw = queue:getBW()
9        if bw == 0 then
10         -- if disconnected
11         -- redirect the request to the edge
12         h:redirect("edge-detector")
13       elseif bw < required then
14         -- if bw is too low
15         -- transform the request to lower-res
16         h:transform("180p")
17       end
18       if bw < required/2 then
19         -- if bw drops well below required
20         -- notify the request source
21         h:notify(bw)
22       end
23     end end end
```

Fig. 3: This simple Lua script for the traffic-monitoring application redirects requests to the edge when the network becomes disconnected, down-samples enqueued requests when bandwidth drops, and invokes a registered callback network conditions change significantly.

iterate over these queues (Line 3) and access various queue properties, such as its length, route, or observed bandwidth (Line 5–8). The queue iterators are also used to apply in place adaptations, such as redirecting messages to another endpoint (Line 12). Other adaptation strategies such as dropping or reordering messages are implemented in a similar fashion. More complex actions, such as transforming messages (Line 16) or notifying the application of metrics (Line 21) can be done through asynchronous callbacks.

Metrics exposed: BumbleBee exports a number of network performance metrics on which to base adaptation decisions; these are summarized in Table I. At the lowest level, BumbleBee exposes TCP metrics such as the congestion window size, number of in-flight packets, and round-trip time (RTT). BumbleBee also exposes the average end-to-end latency for messages in a queue, which Envoy calculates using request and response arrival times. In addition, BumbleBee provides information about how long each messages has spent in a queue through an object-item's age property.

Network bandwidth is a crucial metric for numerous adaptation strategies. BumbleBee does not measure available bandwidth along a physical link, but it calculates the observed bandwidth for messages forwarded from a particular queue. This allows scripts to reason about the observed bandwidth along their path of interest. For example, scripts can detect that a path has been disconnected if its observed bandwidth drops to zero.

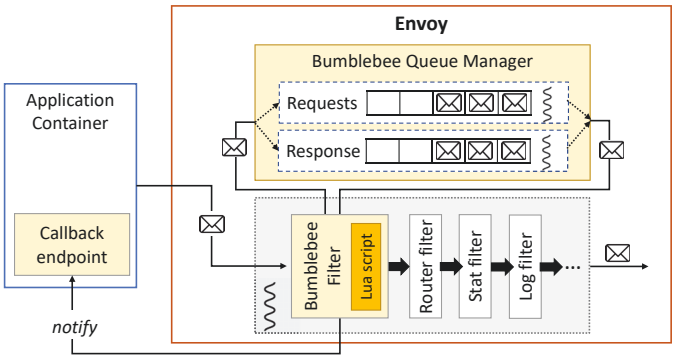


Fig. 4: BumbleBee extends the Envoy sidecar (BumbleBee components marked as yellow).

B. BumbleBee architecture

Figure 4 shows the architecture of BumbleBee. It is situated within an Envoy sidecar, with user-defined adaptation logic executed in the BumbleBee filter. The rest of this section describes the three key components of this architecture: a queue manager that maintains message queues, an in-network scripting facility that executes custom application logic as messages arrive, and a callback mechanism that allows scripts to interact with other parts of an application.

Message queues: BumbleBee represents each (source, destination) pair in a pod as a single, mutable queue, widening the prior interface of stateless message processing. The latter is sufficient for common tasks of a service mesh: load balancing, filtering, etc., but cannot support the reordering pattern needed by many adaptive applications. We add this abstraction with a separate queue manager that runs concurrently with the Envoy worker pool. Worker threads pass messages to the queue manager through the BumbleBee filter. The BumbleBee filter buffers data until a complete network message (e.g., HTTP request/response, Netty [50] message) has been assembled, and then forwards the message to the queue manager. Note that treating messages at higher protocol layers may add some additional latency; as we show in Section IV-D this is modest.

By default the queue manager creates pairs of queues for each endpoint pair, one in each direction, providing a handful of methods to BumbleBee filters within the worker threads. The application can optionally request finer-grained division of queues on a per-pod basis. For example, in its orchestration configuration an application may name pods containing an object-detector running on the cloud "cloud-object-detector.local." It can then instruct BumbleBee to create queues in each pod's sidecar for handling requests to those specific pods. Individual filters are invoked as messages arrive on inbound queues, and pass messages after processing to the corresponding outbound queue. Because operations are non-blocking, we use a simple per-queue locking scheme to synchronize access.

In addition to events based on message arrival, the manager periodically receives timer events that implement a token-bucket algorithm for outbound messages. When the manager has accumulated enough tokens, messages are forwarded to

Context	Interface	Description	Returns
Queue	length()	returns number of messages in a queue, useful to approximate queuing delay.	queue length
	avgLatency()	returns weighted moving average of end-to-end latency of messages (delta between request & response)	average latency
	observedBW()	returns observed bandwidth allocated to the queue–the rate of the queue sending data.	observed bandwidth
	TCPMetrics(m)	retrieves the TCP metrics (e.g., mean RTT) at the queue level.	TCP metric
	messages()	for-loop entry to iterate over messages in the queue.	message object
Message	size()	returns the size of the message's current payload.	size of payload
	age()	get the age, i.e., how long the message has been in the queue, in ms resolution.	age of message
	TCPMetrics(m)	retrieves the TCP metrics (e.g., mean RTT) at the message/request level.	TCP metric
	dst()	returns the current destination of the message.	message destination
	header()	returns the message's header.	message header
	bytes(i, j)	returns data from `i` to `j` of payload of the current message in raw binary format.	raw payload
	redirect(dst)	redirect the message to a new destination (dst).	
	transform(args)	asynchronously transform a message's payload by forwarding to a registered endpoint.	
	drop()	drops the current message from the queue. The function does not guarantee successful operation (e.g., already transmitted in the middle of dropping). If successful, returns the updated queue length, otherwise, returns the old queue length.	new queue length
	insert(msg)	inserts a new message `msg` after the current message in the queue. If successful, returns the updated queue length, otherwise, returns the old queue length.	new queue length
	moveToFront()	move the message to front of the queue.	
	moveToBack()	move the message to end of the queue.	
Callback	notify(metrics)	asynchronously send registered endpoints a metrics string.	

TABLE I: BumbleBee interface for in-network scripting.

Envoy's event dispatcher. To minimize overhead, timers scale the token refill rate and are only active when the queue has pending messages.

Scripting facility: BumbleBee applies user-defined adaptation logic to the queues maintained by the queue manager. When a worker thread loads a BumbleBee filter, the filter reads the appropriate script from the orchestration configuration and launches it within a Lua runtime, a feature natively supported by Envoy. These scripts are executed only as messages arrive; we have chosen not to also add timer events. This ties adaptation agility to message arrival rate; a limitation we have not found burdensome in our limited experience so far.

External callbacks: BumbleBee's scripting environment allows applications to perform simple processing on enqueued messages: drop, redirect, or reorder. However, many applications can benefit from richer interactions between adaptive scripts and the rest of the application. For example, an application might benefit from in-network observation, providing early detection of bandwidth or latency changes. Likewise, an application may want to transform message payloads in ways that are too complex for a lightweight Lua runtime. For example, a video streaming application may want to downsample video chunks ahead of network constructions to prevent head-of-line blocking. To support this kind of functionality, BumbleBee allows scripts to make asynchronous callbacks.

There are two forms such callbacks might take. The first (and simpler) one is used to *notify* external endpoints of events within an adaptation script. An application's orchestration configuration can bind a list of RESTful endpoints to particular notifications within a script, taking a string as an argument. On invocation, the Lua runtime generates asynchronous HTTP calls with the string argument to any endpoints listed in the orchestration configuration. This exposes information from lower layers, and so should be used in the rare cases when an application benefits significantly from such feedback.

The second form is used to *transform* message contents. As with notifications, applications bind invocations to a RESTful endpoint through their orchestration configuration. When a script invokes transformation on an enqueued message, the Lua runtime marks the entry asynchronously forwards the message payload to the registered endpoint for transformation. The result is returned and substitutes for the original message. To avoid blocking on this operation, pending messages are marked in progress, and subsequent messages can be sent in the interim. Transformations are typically used for complex computations that should not take place in the critical path of the Envoy sidecar. Transformations may optionally access other resources–such as an external database–that are not possible within a sidecar limited only to the standard libraries.

IV. Evaluation

To evaluate BumbleBee, we seek answers to the following questions:

- Does BumbleBee enable beneficial adaptation strategies?
- How difficult is writing adaptation strategies in BumbleBee?
- How much overhead does BumbleBee add to Envoy?

To answer the first three questions we use our BumbleBee prototype to investigate adaptation strategies for three case-study applications. First, we use BumbleBee to help a distributed, vehicular-traffic monitoring application that adapts the quality of its object detection to changing network conditions. Second, we use BumbleBee to help a stream-processing application intelligently shed requests under bursty workloads. Finally, we use BumbleBee to help a live video-streaming service to reduce stalled playback while maintaining acceptable video resolution. To answer the last question, we run wrk2 [75] micro-benchmarks to measure how BumbleBee affects request latency compared to Envoy.

We run these workloads with BumbleBee using Istio 1.4.3, Envoy 1.13.0, and clusters of virtual machines managed by Azure Kubernetes Service (AKS) 1.18.14.

A. *Case study: traffic monitoring*

Our first case study is an emulated smart-city application that streams roadside video to machine-learning (ML) models. The ML models forward a detected vehicle's bounding box and confidence level to one or more traffic-light controllers. The controllers filter bounding boxes with confidence levels below a threshold (e.g., 50%). The controllers use vehicle counts and locations to monitor and schedule traffic, such as reducing the time between green and red lights when road congestion is high.

Traffic monitoring is representative of many edge computing applications [55]. The input sensors (e.g., roadside cameras) and controllers (e.g., traffic controllers) are co-located on the edge with a distributed computing pipeline between them. This pipeline must process sensor data fast enough for the controllers to respond to changes in the physical environment, and the application must operate even when network conditions are poor.

The ML pipeline can be instantiated along two paths: embedded in a resource-rich cloud environment or a lightweight edge environment. The cloud offers powerful machines and can support sophisticated and accurate ML models, whereas the edge can run a limited number of less accurate models. The application prefers results from the cloud models, and it will send frames to the cloud as long as network conditions allow it.

Detection accuracy is a key measure of fidelity for traffic monitoring. Accuracy is highest when the network allows the application to stream high-resolution frames to the cloud, but as network conditions change, the application can adapt the video stream's quality by changing frame resolution or frame rate. Low-quality streams diminish model accuracy, and high-quality streams improve accuracy. The application runs at lowest fidelity when it is disconnected from the cloud. During disconnections, the application must redirect video frames to its lightweight edge models, sacrificing accuracy for availability.

Figure 3 from Section III shows a BumbleBee script that implements these trade-offs. The script iterates over an egress request queue looking for entries destined for a cloud object-detector. When bandwidth drops to zero, BumbleBee redirects requests to the edge object-detector. If bandwidth falls below a threshold, BumbleBee forwards requests to the application's transform service, which reduces frames' resolution to 180p (320x180). And if bandwidth falls well below what is required, BumbleBee notifies the sender so that it can start to send lower-resolution frames instead of relying on BumbleBee to do so.

Major cloud providers like AWS [1], [4], Azure [6], [52], and Google Cloud [26], [27] all suffer significant outages, and recent studies show that network conditions between the edge and cloud can be turbulent [13], [55], [77]. To understand how our traffic-monitoring application behaves when edge-to-cloud connectivity is poor, we run experiments with disconnections and constricted bandwidth between the edge pods and cloud pods. Note that for our experiments we logically divide cluster nodes between the edge and cloud, but the underlying physical machines and network are entirely in Azure. Our Kubernetes cluster contains virtual machines with four vCPUs, 16 GB RAM, and a 32 GB SSD. The cluster also includes two GPU nodes with an Nvidia Tesla K80 GPU, six vCPUs, 56 GB RAM, and a 340 GB SSD. We simulate a roadside camera by streaming a highway-traffic recording from Bangkok, Thailand [11]. We use YOLOv3 as our cloud object-detection model and TinyYOLO as our edge model. Both models are trained with the COCO dataset [45], which is designed to detect vehicles and passengers.

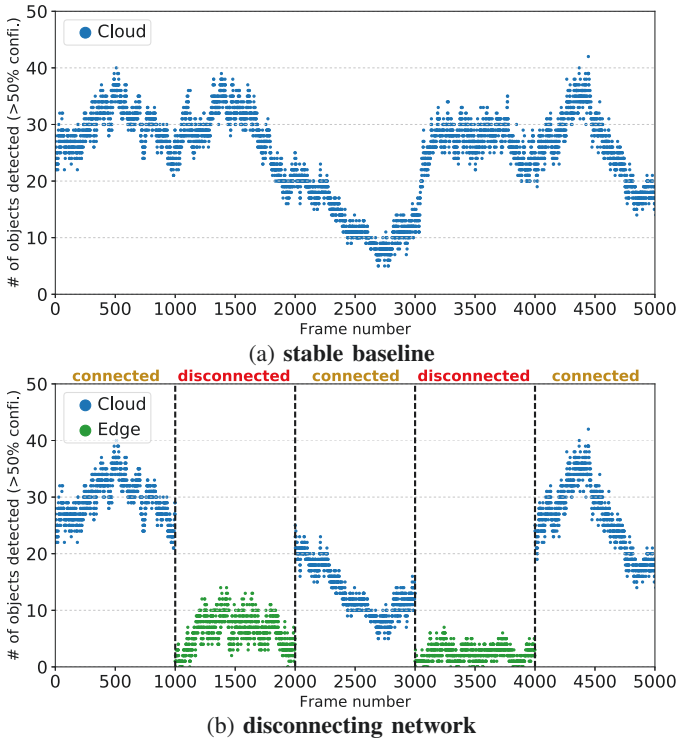


Fig. 5: The number of detected vehicles with greater than 50% confidence with (a) stable network conditions and the workload running fully in the cloud, and (b) network disconnections that shift the workload to the edge using BumbleBee.

To evaluate if the application benefits from BumbleBee, we measure the number of detected vehicles and end-to-end detection latency. The former metric influences how well the application controls traffic, and the latter influences how quickly the light controller responds to traffic changes.

To characterize our traffic-monitoring application without BumbleBee, we first capture the baseline object-detection accuracy of streaming 360p (640x360) video at 15 fps when fully connected to the cloud. Figure 5a shows the number of detected vehicles over time with a confidence threshold above 50%. The YOLOv3 model in the cloud consistently detects

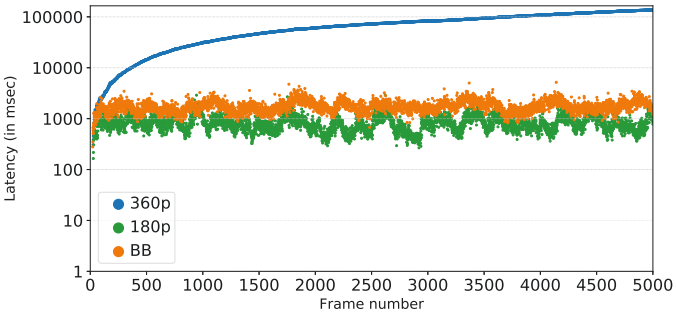


Fig. 6: When edge-to-cloud bandwidth is 15 Mbps, sending 360p frames leads to head-of-line blocking and exponentially increasing detection latency. Sending 180p frames reduces median latency to 815 ms with no head-of-line blocking. BumbleBee (BB) allows the application to selectively downsample frames to balance latency (median latency of 1700ms) and detection accuracy.

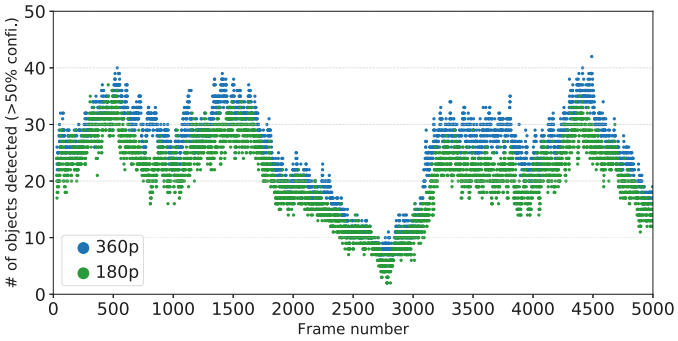


Fig. 7: The cloud object detector identifies more vehicles with confidence greater than 50% in 360p frames than in 180p frames.

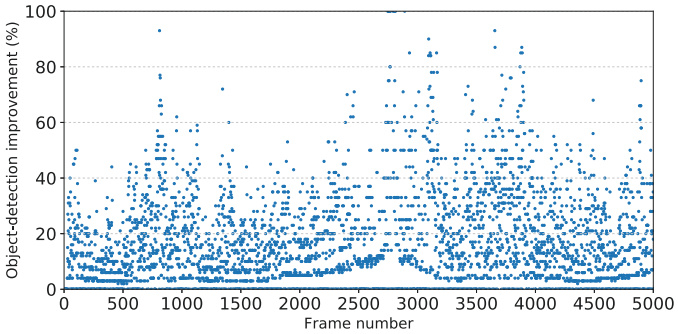


Fig. 8: BumbleBee enables the traffic-monitoring application to send 360p frames when possible and avoid head-of-line-blocking by selectively downsampling frames to 180p. Each blue data point represents the percentage of additional objects that the BumbleBee-enabled application detects in a frame compared to sending all 180p frames. The 3444 frames (out of 5000) are downsampled to avoid exponential queuing delays. These frames gain zero percent improvement.

between 10 and 40 vehicles.

To simulate a disconnected edge site, we run the application under BumbleBee and partition the edge and cloud pods after 1000 and 3000 frames so that the cloud object-detector is unreachable. We heal the network between frames 2000 and 3000. Loading tensor-flow models can be slow, so BumbleBee pre-loads the edge detector at the beginning of the experiment. Figure 5b shows the detected cars drop during disconnection, because the application switches to TinyYOLO on the edge.

There is a delay between when a disconnection occurs and when BumbleBee detects the disconnection. In our experiment, five frames stall before BumbleBee detects that bandwidth is zero. Recall that our video streams at 15 fps, and so requests arrive every 67 ms. Thus, the first request sent after the disconnection experiences an approximately 350 ms of additional delay before BumbleBee redirects it to the edge. This is because four cloud-bound requests arrive after the first post-disconnection request but before BumbleBee detects the disconnection. When the sixth post-disconnection request arrives, BumbleBee has detected the disconnection and responds by redirecting all cloud-bound requests to the edge. Between disconnections, the application matches baseline detection accuracy. These results show that with BumbleBee, the application can continue to operate, albeit in a degraded mode, when the cloud is unavailable.

We also want to understand if the application benefits from adapting to network changes that are less dramatic than a disconnection. Recall that end-to-end latency is a critical application metric. When disconnected, the weaker edge detector processes 360p frames 33% faster than the cloud detector using equivalent hardware. However, when bandwidth to the cloud drops, sending 360p frames can cause exponentially increasing queuing delay. To demonstrate, we restrict edge-to-cloud bandwidth to 15 Mbps and repeat the traffic-monitoring experiment twice, first sending 360p frames and then sending 180p frames. Frames are full-color, JPEG-compressed images. Figure 6 shows the results. When the application streams 360p frames (the blue line), the latency rises exponentially, but the median latency of streaming 180p frames is 772 ms.

However, lower resolution frames reduce detection accuracy. Figure 7 shows the number of objects the application detects with confidence greater than 50% for 360p and 180p frames. Note that the blue dots are identical to those in Figure 5a. 360p frames allow the cloud model to consistently detect more objects than the 180p stream, often significantly so. These results suggest that the traffic-monitoring application could benefit from selective adaptation by downsampling frames that cause queuing delay, and transmitting the remaining frames intact.

To confirm our hypothesis, we repeat our limited-bandwidth experiment using BumbleBee. The application sends 360p frames, and BumbleBee selectively downsamples frames that cause queuing delay. Figure 8 shows the percent improvement of the object detector with BumbleBee's selective downsampling enabled compared to always sending 180p frames. When BumbleBee downsamples a frame to 180p, the improvement percentage is zero. Overall, BumbleBee downsamples 3444 frames and leaves 1556 intact. Furthermore, the graph shows

that selectively downsampling provides much better detection accuracy than always downsampling. Combined with the median latency of 1700ms in Figure 6, these results show that BumbleBee allows the traffic-monitoring application to find a good balance between detection accuracy and latency using the simple script in Figure 3.

To summarize, the results show that our traffic-monitoring application benefits from BumbleBee in two ways. First, the application operates when disconnected from the cloud by redirecting requests to a weaker edge object-detector. Second, when network bandwidth constricts, the application selectively downsamples frames to balance end-to-end latency and detection accuracy. We also show that the adaptation strategies responsible for these benefits can be concisely expressed by the script in Figure 3.

B. Case-study: stream processing

Our second case study is the Yahoo! stream-processing benchmark [14] that counts ad views from an input stream of ad impressions, i.e., clicks, purchases, and views. The benchmark is widely used [33], [46], [70], [78], because it mimics in-production workloads and business logic. The first stage reads and parses impression data, the second stage filters out non-view events, and the final stage stores aggregate view counts over 10 s sliding windows. Impression counts help ad services bill customers and select the next ads to display. In the latter case, *timeliness* (meeting a latency deadline) is more important than *completeness* (fully processing every input), and many practitioner testimonials [21], [22], [69] emphasize the importance of timeliness.

By default, the Yahoo! benchmark generates emulated impressions at a constant rate, but real-world rates can be bursty. Bursts may be problematic for applications with timeliness requirements, because practitioners often statically allocate resources and must restart pipelines to scale dynamically [70]. Over-provisioning is not always possible, and unexpected bursts can rapidly increase end-to-end latency as applications fall behind processing every message.

Load-shedding [64], [65], [76] is a common way to adapt to such bursts. Shedding trades completeness for timeliness by dropping less important inputs to free resources and improve the number of deadlines met. Today this adaptation strategy can only be implemented by modifying an application's internals, but BumbleBee can intelligently shed load for unmodified applications.

To characterize how effectively BumbleBee helps the Yahoo! benchmark improve timeliness, we orchestrate the benchmark with Kubernetes by placing a containerized Apache Flink [12] worker in a pod. A worker pod can execute any stage and can pass inter-stage data within the same pod. Each Kubernetes node hosts one pod and is a virtual machine with two vCPUs and 8 GB RAM, connected by an underlying network provisioned at 1 Gbps. The benchmark polls external Kafka brokers for input events and stores results in an external Redis database. 10 s sliding windows are too coarse to properly measure the impact of bursts on timeliness, so we add a small amount of instrumentation to aggregate over 1 s sliding windows.

```
1  filt_thrd = 0.5 --- filtering threshold in sec
2  late_thrd = 1.0 --- lateness threshold in sec
3  function envoy_on_response(h)
4   queues = h:Queues()
5   for queue in queues:getQueue() do
6     for msg in queue:messages() do
7       json = msg:json()
8       if queue:avgLatency() > filt_thrd then
9         event_type = json:getString("event_type")
10        if event_type ~= "view" then
11          msg:drop()  --- pre-emptively filter
12        end
13      end
14
15      event_time = json:getNum("event_time")
16      age = h:epoch() - event_time
17      if age > late_thrd then
18        msg:drop()    --- drop late msgs
19      end
20    end end end
```

Fig. 9: This BumbleBee script pre-emptively filters messages and drops late messages to save inter-pod bandwidth when it detects latency in the pipeline.

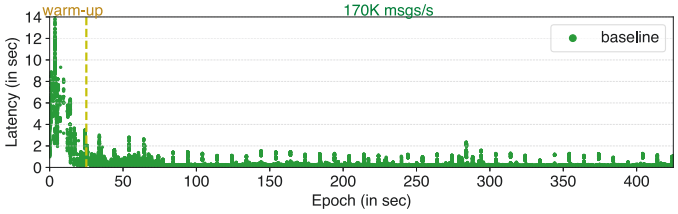


Fig. 10: Our stream-processing application processes input messages mostly under 2 s latency after a short warm-up period, when 170k input messages are streamed per second.

Figure 9 shows a BumbleBee script that uses custom message-dropping logic to implement two forms of load shedding: pre-emptive filtering and dropping late messages. Recall that the benchmark filters out click and purchase events in its second stage. Under BumbleBee, if latency increases, the benchmark pre-emptively filters non-view events before the second stage (lines 8-13). The script also drops view events if they are unlikely to meet their deadline (lines 17-19). Both adaptations free resources as the script detects latency in the pipeline.

Our baseline benchmark configuration runs under Kubernetes, without an Envoy sidecar or BumbleBee. We first run the baseline configuration with a constant, baseline load of 170k events per second. To characterize latency, we sample the end-to-end latency of the last event included in the benchmark's 1 s aggregation window. Figure 10 shows how the latency of these sampled events change over time. Latency for the first 25 s is highly variable as the benchmark warms up. After the warmup, sampled latency is largely under 2 s. This is expected since we provisioned enough compute and network resources to process every event within 2 s.

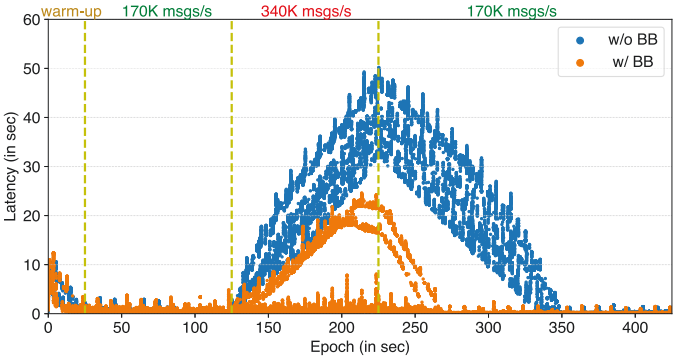


Fig. 11: Temporal 2x input load spikes leads the application to experience high latency for 1.5 times longer than the spike duration even after the input load returns back to the previous level. The BumbleBee-enabled application takes less than 30 s to bring latency back to the previous level.

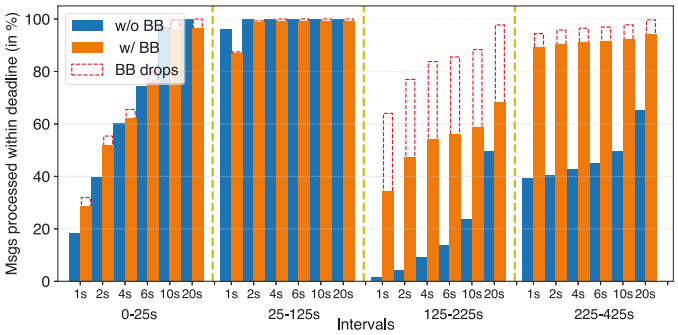


Fig. 12: When the input load increases above expected level that operators have projected and provisioned resources accordingly, the application hardly processes messages within a deadline. The consequence continues to stay longer than the ramp-up period.

We next run an experiment with variable load: first 170k events per second for 125 s, followed by a burst of 340k events per second for 100s, followed by a return to 170k events per second for 200s. Figure 11 shows sampled latency for the baseline benchmark (w/o BB) and the benchmark with BumbleBee (w/ BB) under variable load. During the burst, BumbleBee drops over 29% of all events, and after the burst, BumbleBee drops less than 6% of events. Compared to the baseline, BumbleBee's custom dropping policy significantly improves sampled latency and time to recovery. Excluding warmup, BumbleBee allows nearly 74% of sampled views to be processed within 2 s, whereas the baseline benchmark allows only 44%. In addition, BumbleBee returns the benchmark to steady state less than 50 s after the burst ends; without BumbleBee, it returns to steady state after 125 s.

A limitation of the current BumbleBee implementation causes the two arcs in Figure 11 that peak at 20 s and 25 s sampled latency. BumbleBee intercepts only inter-pod communication, but benchmark pods contains workers for all stages. Thus, sometimes the benchmark transfers data between stage workers residing in the same pod, i.e., over local Unix sockets on which BumbleBee cannot interpose. This phenomenon is an artifact of the Yahoo! benchmark's design and would not be an issue for applications that separate each stage into a dedicated tier of pods.

Figure 12 highlights how BumbleBee impacts meeting deadlines of 1-20 s during the warmup, baseline-load, bursty-load, and second-baseline intervals. As expected, longer deadlines (e.g., 20 s) are met more often than shorter ones (e.g., 1 s) with and without BumbleBee. There is also little difference between the two configurations during the initial warmup and baseline intervals. However, BumbleBee provides substantial benefit during the bursty interval, allowing the benchmark to meet nearly 23x more 1 s deadlines and 37.8% more 20 s deadlines than without BumbleBee. BumbleBee provides substantial benefits when load returns to normal, allowing the benchmark to meet over 90% of its deadlines, regardless of length. In contrast, the baseline benchmark only meets less than 40% of its 1 s deadlines and 65% of its 20 s deadlines. This is due to the baseline benchmark's emphasis on completeness, and having to work through its backlog of enqueued events even after load has returned to normal.

C. Case study: video streaming

For our final case study, we evaluate an HTTP Live Streaming (HLS) service with an Nginx server and HLS.js client [32]. At runtime, the server partitions an input live stream into a rolling sequence of self-descriptive, fixed-length MPEG-TS chunks at several resolutions. When a chunk can be downloaded, the server updates an HLS manifest file to announce its availability and resolution. The HLS client is responsible for all adaptation logic and periodically polls the manifest to learn when the newest chunk is ready. After reading the manifest, the client predicts the time to download the next chunk at the available resolutions. These predictions are based on the chunks' sizes and a bandwidth estimate calculated over a sliding window of prior downloads.

The client's competing objectives are continuous video playback and high video quality. Stalling occurs when the client's playback buffer is empty, which is far worse for the user experience than temporary drops in video quality [20]. For example, if bandwidth drops in the middle of downloading a chunk, the client's playback buffer may drain before the transfer completes. This is common when clients react too slowly to abrupt bandwidth drops and high variability [47], [77].

Prior solutions to this problem rely on either server [47] or client [77] modifications. Modifying a server without the clean separation provided by BumbleBee requires either building from scratch or understanding an existing codebase and continuously merging with external updates. Furthermore, client agnosticism is critical for open protocols like HLS, because service providers cannot dictate which of the many players a client may use [68]. Fortunately, BumbleBee can transparently

```
1  function envoy_on_request(h)
2    hdr = h:header()
3    bw = hdr:get("bw-est")
4    curr, chunk = hdr:get("path")
5    -- use bw estimate to choose a chunk
6    pred = find_resolution(bw)
7    if pred < curr then
8      -- downsamples
9      hdr:replace("path", pred.. "/".. chunk)
10   elseif pred > curr * 2 then
11     -- upsamples
12     hdr:replace("path", pred.. "/".. chunk)
13   end end
```

Fig. 13: This BumbleBee script for the video streaming application predicts appropriate resolution to transmit based on the most recent bandwidth measurement and distribution of chunk sizes. When the script disagrees with the client, it overwrites the `path` of chunk's resolution to increase/decrease resolutions. Note that the script is conservative about upsampling to avoid potential stalls.

apply a variety of adaptation strategies to correct an HLS client's bandwidth mis-predictions.

The BumbleBee script in Figure 13 illustrates such a strategy. The script adapts to sudden bandwidth changes faster than an unmodified HLS.js client by only considering the most recent chunk transfer rather than a sliding window over several transfers. Based on this bandwidth estimate, the script chooses among available chunk resolutions. If the requested resolution could cause a stall (line 6-9), BumbleBee modifies the path field of the HTTP header so that it refers to a lower-resolution chunk. However, if the client requests a resolution that could under-utilize the available bandwidth (line 10), BumbleBee swaps in a higher-resolution chunk path. BumbleBee's bandwidth estimate requires Envoy modifications to monitor low-level transfer progress or a service provider to place a middlebox between the client and server. For the purposes of our experiments, we emulate the latter by co-locating a proxy with the client and configuring the client to direct its requests through the proxy. Future versions of BumbleBee will include the necessary Envoy modifications.

We first evaluate the video-streaming service with two synthetic bandwidth changes: a sudden drop and recovery, and a gradual drop and recovery. These changes highlight the trade-offs of reacting more quickly than the HLS client's strategy. In addition, to evaluate the efficacy of BumbleBee in real-world scenarios, we analyze network-condition logs of Puffer [77] clients watching live video streams. We limit our experiments to traces that cause stalls of more than 100 s, and from these traces replay estimated instantaneous bandwidth conditions. We replay the first ten minutes of each trace.

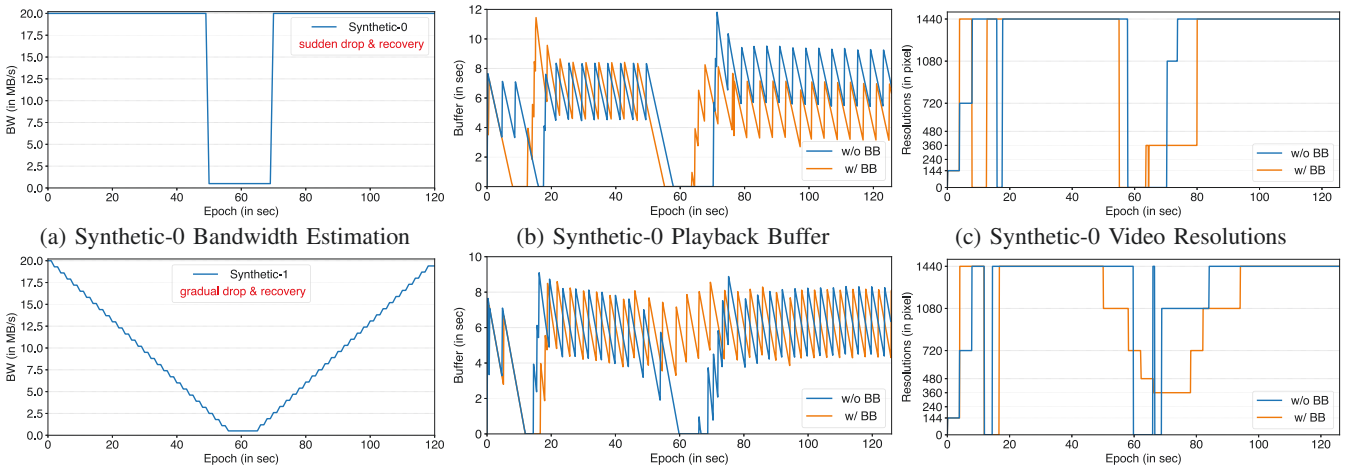
For all experiments we run the Nginx server under Kubernetes in a dedicated virtual machine with an Nvidia V100 GPU, 6 vCPUs, and 112 GB of RAM. For the client, the HLS.js player in the same data center as the Kubernetes cluster but in a separate virtual machine with sufficient underlying bandwidth between the two. We use Linux `TC` to replay bandwidth traces at the client side, and we use the default player configuration unless noted. Each video chunk is four-seconds long.

We characterize streaming with and without BumbleBee with two metrics: playback buffer and video resolution. The playback buffer is the seconds of video that a client can play without receiving new data from the server. When the buffer reaches zero, the video stalls. Resolution represents video quality. Buffer and resolution can be traded off. In the extremes, sending only low-resolution chunks minimizes quality but maximizes buffer, and sending only high-resolution chunks maximizes quality but minimizes buffer. Because stalls are so painful [20], BumbleBee wants to offer acceptable quality with minimum stalls.

Figure 14a shows the first synthetic trace: a sharp bandwidth drop for 20 s and fast recovery. Figures 14b and 14c show the clients' playback buffer levels and displayed resolutions over the course of the trace, respectively. The client under both configurations stalls as it calibrates its bandwidth estimates. The client under both configurations also stalls when bandwidth drops. However, under BumbleBee the client adapts to the drop and rebuilds its playback buffer faster than without BumbleBee. Overall, BumbleBee reduces stalling from 13 s to 9 s, a 32% improvement. As Figure 14c shows this is possible because under BumbleBee the client reduces its resolution to 360p near 65 s, whereas without BumbleBee the client continues to download 1080p chunks.

Figure 14d shows the second synthetic trace: gradual bandwidth decrease and recovery, each over 50 s. We hypothesized that BumbleBee would offer little benefit on this trace, anticipating that the client's default estimates would closely track the gradual changes. Surprisingly, Figure 14e shows that BumbleBee reduces post-calibration stalling from 11 s to 5 s, a 55% improvement. Figure 14f shows that without BumbleBee the client fails to adapt to decreasing bandwidth, continuing to fetch 1440p chunks. In comparison, BumbleBee reduces resolution in a step-wise fashion and eliminates all stalling in the valley.

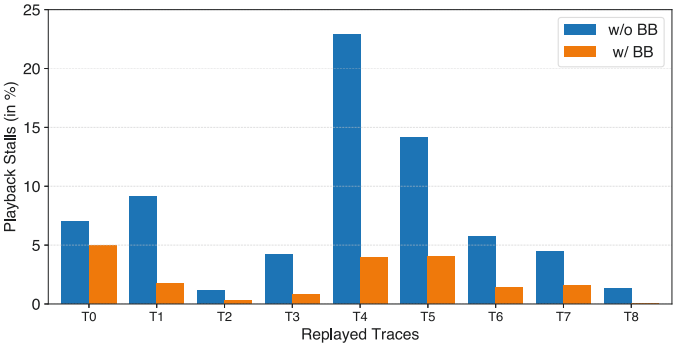
We repeat the experiments with nine Puffer traces. Figure 15a summarizes the percentage of total stall time that a client experiences during each trace, with and without BumbleBee. The client with BumbleBee stalls at the most 5% of the total duration, and the client without BumbleBee stalls 22% of the time, a 77% improvement. Figure 15b shows box plots of playback resolution, including mean and median. Note that in trace T2, which exhibits the least stalling without BumbleBee, the HLS.js client achieves higher resolutions than with BumbleBee albeit with some additional stalling. From the logs, we find that BumbleBee's script is too cautious about sending higher resolutions that could clog the connection during T2. This matches our expectation that quickly reacting to network changes to aggressively avoid stalling can lead to worse bandwidth utilization.



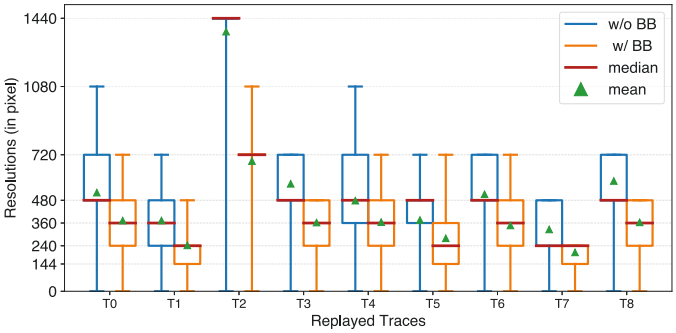
(a) Synthetic-0 Bandwidth Estimation (b) Synthetic-0 Playback Buffer (c) Synthetic-0 Video Resolutions

(d) Synthetic-1 Bandwidth Estimation (e) Synthetic-1 Playback Buffer (f) Synthetic-1 Video Resolutions

Fig. 14: BumbleBee helps the live video streaming application to adapt quickly and cautiously. Figures in the first column show the bandwidth estimates for both traces. Figures in the second column show how the client's playback buffer changes during the trace. Figures in the last column demonstrates fast and agile adaptations by BumbleBee's script.



(a) Across all traces, playback stalls are significantly less with BumbleBee than without.



(b) To eliminate stalls, BumbleBee sacrifices some video resolution.

Fig. 15: Experiments with the nine Puffer traces with the most stalls show how BumbleBee helps the live video streaming application to reduce stalls while maintaining acceptable video resolution.

D. Latency micro-benchmarks

To evaluate the overhead imposed by BumbleBee compared to Envoy, we measure end-to-end latency using the HTTP benchmarking tool wrk2 [75]. The tool generates HTTP requests at a constant rate and outputs a latency distribution. We configure wrk2 to generate 500 requests per second with 1000 concurrent connections over five one-minutes runs.

For our experiments, we create a client pod that runs wrk2 and assign it to a GPU node. We also create a server pod running the Nginx web server under default settings on a normal node. Both pods contain an Envoy sidecar with access to two cores. We measure the latency distribution under four client configurations: (1) Envoy without BumbleBee, (2) BumbleBee with no Lua script, (3) BumbleBee with a simple queue-iteration script, and (4) BumbleBee with a simple LIFO (Last In First Out) script. The first configuration serves as baselines for understanding BumbleBee's scripting overhead. Note that all configurations with BumbleBee move messages from the BumbleBee filter to the queue manager.

We use a simple LIFO and queue-iteration scripts used in the experiments. BumbleBee's queues are internally implemented as doubly-linked lists, which makes LIFO reordering relatively inexpensive. However, iterating over the queue could be slow for two reasons. First, BumbleBee uses a per-queue locking scheme that ensures only one script can execute at a time. Second, the Lua runtime creates a new stack and object bindings each time the iteration script runs. These startup costs are drawbacks of using a scripting language instead of binary executables or bytecodes like WebAssembly.

To test our hypothesis, we run wrk2 five times with each client configuration. Figure 16 shows the latency distributions at the 50th, 75th, 90th, and 99th percentiles on the X-axis. Up to the 75th percentile, the latency for all BumbleBee configurations is very close to Envoy, between 6.5% to 12% extra overhead, where the absolute value for the latency overhead is between $0.15ms$ and $0.35ms$. However, the cost of iterating over the queue is apparent at the very tail of the distribution. For example, at the 90th percentile, the iteration script's latency is 23% more than Envoy's, and at the 99th

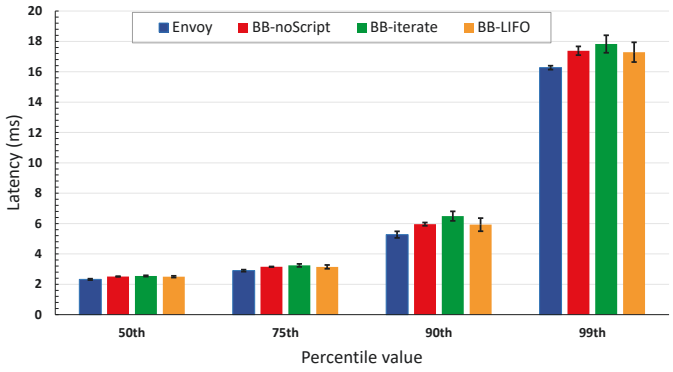


Fig. 16: Micro-benchmarks with wrk2 show that BumbleBee adds no additional overhead compared to Envoy up the 99th percentile. However, the additional cost of iterating over a message queue becomes apparent at the very tail of distribution.

percentile it is 9.5% more.

V. Related work

Adaptation in mobile computing: Resources in mobile computing are highly constrained as opposed to a data-center environment. Prior adaptation systems [23], [54] trade application fidelity for various metrics. Similar to BumbleBee's callback functionality, Odyssey [54] creates a collaborative adaptation solution that notifies applications to adapt their fidelity. On-demand distillation [23] performs "on-the-fly" adaptive transcoding of web contents based on the client's bandwidth, similar to BumbleBee's dynamic transformation. However, these systems do no expose control over the enqueued data.

Many others [7], [15], [17] integrate adaptation logic for better use of computation resources. Cyber foraging [7] is a runtime framework that allows developers to write and deploy complex adaptation tactics. MAUI [17] and CloneCloud [15] partition application code, either with developer-defined annotations (MAUI) or through static analysis (CloneCloud). Then, they adaptively offload partitions between local execution (on the mobile device) to remote execution. BumbleBee can be thought of as an extension to these systems where it can redirect the offloading traffic based on runtime variables such as network bandwidth.

Adaptation in video streaming: Video streaming [34], [38], [48], [56], [62], [77] is another domain that employs various adaptation strategies to improve video watching experience. A few recent works [47], [77] propose video streaming servers that adaptively select the best bit-rate by using machine learning to predict the bandwidth or transmission time. Others have developed video clients to adapt to network conditions changes for fairness [38] and stability [34], to minimize rebuffering [62], and to handle unexpected network outage [56]. While the individual solutions vary, these solutions can easily be reimplemented in BumbleBee and leverage the low-level networking metrics and control available by BumbleBee to achieve improved performance (as shown in Section IV-C).

Other Adaptations: Odessa [57] is an adaptive runtime for partitioning and executing computer vision application remotely. The runtime balances the level of pipelining and data-parallelism to achieve low latency under variable network conditions. Kahawai [18] is a system for cloud gaming where clients with modest local GPUs collaborate with powerful cloud servers to generate high-fidelity frames. Kahawai adapts to network changes by adjusting the fidelity and frame rate of frames. Outatime [41] is a speculative execution system for cloud gaming where thin-clients send input and servers at the cloud render speculative frames of future possible outcomes while adapting to network tail latencies. These systems can leverage the scripting interface and in-network processing capabilities of BumbleBee to improve or simplify their adaptation strategy.

In-network Processing: The concept of in-network processing has been proposed over two decades ago where custom in-network applications are deployed at the router to provide additional functionalities, e.g., webpage caching [67]. Recent developments in networking hardware (e.g., smart NIC, FPGA) have led to revisiting the idea of in-network processing. Flexible programming languages such as P4 [9] have emerged to simplify the development of in-network processing applications. As a result, many [29], [42], [43], [63] have explored using in-network processing for wide variety of use cases such as improving consensus protocols (NOPaxos [43]), faster transactions (Eris [42]), network telemetry (Sonata [29]), or improving network functionalities, e.g., DNS and NAT (Emu [63]). Along the lines of these work, BumbleBee allows in-network processing of custom adaptation logic but for containerized environments such as Kubernetes.

VI. Conclusions

In this paper we describe BumbleBee, a platform supporting application-aware adaptation that is integrated with the orchestration and service mesh mechanisms that support container-based microservices. This is done by judiciously widening the in-network interface in two ways. From above, applications supply simple scripts that describe adaptive logic. From below, service mesh sidecars expose the queue of pending messages so that these scripts can drop, reorder, redirect, or transform those messages. Experiments with a BumbleBee prototype demonstrate the benefits of our approach: (1) by using BumbleBee, ML applications at the edge can utilize cloud resources when available and operate without interruption when disconnected, (2) BumbleBee increases the number of deadlines met between 23x and 37.8% on the Yahoo! stream-processing benchmark, and (3) BumbleBee reduces stalled playback by 77% during HLS video streaming under real-world network conditions.

Acknowledgments

We thank our reviewers for their thoughtful comments. This work has been partially supported by the National Science Foundation under grant CNS-1717064.

REFERENCES

[1] Amazon's massive AWS outage was caused by human error. https://www.vox.com/2017/3/2/14792636/amazon-aws-internet-outage-cause-human-error-incorrect-command. Last accessed May 2021.

[2] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

[3] Hybrid cloud with aws. https://aws.amazon.com/hybrid/. Last accessed September 2021.

[4] AWS left reeling after eight-hour DDoS. https://www.infosecurity-magazine.com/news/aws-customers-hit-by-eighthour-ddos/. Last accessed May 2021.

[5] Azure arc overview. https://docs.microsoft.com/en-us/azure/azure-arc/. Last accessed September 2021.

[6] Azure global outage: Our DNS update mangled domain records, says Microsoft. https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/. Last accessed May 2021.

[7] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services (MobiSys)*, pages 272–285, 2007.

[8] Gaurav Banga, Peter Druschel, and Jeffry C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (SIGCOMM)*, 44(3):87–95, 2014.

[10] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Denver, CO, June 2016.

[11] Panasonic Business. 4K camera road in Thailand No 2. https://www.youtube.com/watch?v=F4bICvLY024. Last accessed May 2021.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[13] Yang Chen, Jens O Berg, Mostafa Ammar, and Ellen Zegura. Evaluation of data communication opportunities from oil field locations at remote areas. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC)*, pages 117–126, 2011.

[14] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *Proceedings of the IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792, 2016.

[15] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems (EuroSys)*, pages 301–314, 2011.

[16] Cilium. https://cilium.io.

[17] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys)*, pages 49–62, 2010.

[18] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai: High-quality mobile gaming using GPU offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 121–135, 2015.

[19] Luca De Cicco, Saverio Mascolo, and Vittorio Palmisano. Skype video responsiveness to bandwidth variations. In *Proceedings of the 18th international workshop on network and operating systems support for digital audio and video*, pages 81–86, 2008.

[20] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. *ACM SIGCOMM Computer Communication Review*, 41(4):362–373, 2011.

[21] Complex event generation for business process monitoring using Apache Flink. https://engineering.zalando.com/posts/2017/07/complex-event-generation-for-business-process-monitoring-using-apache-flink.html. Last accessed March 2021.

[22] Real-time experiment analytics at Pinterest using Apache Flink. https://medium.com/pinterest-engineering/real-time-experiment-analytics-at-pinterest-using-apache-flink-841c8df98dc2. Last accessed March 2021.

[23] Armando Fox, Steven D Gribble, Eric A Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 160–170, 1996.

[24] Gartner says four trends are shaping the future of public cloud. https://www.gartner.com/en/newsroom/press-releases 2021-08-02-gartner-says-four-trends-are-shaping-the-future-of-public-cloud. Last accessed September 2021.

[25] Anthos. https://cloud.google.com/anthos. Last accessed September 2021.

[26] Google Cloud in major global outage: Numerous services fail. https://www.cbronline.com/news/google-cloud-down. Last accessed May 2021.

[27] Google Cloud Status dashboard. https://status.cloud.google.com/summary. Last accessed May 2020.

[28] Ben Greenstein, Christopher Mar, Alex Pesterev, Shahin Farshchi, Eddie Kohler, Jack Judy, and Deborah Estrin. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys)*, pages 279–292, 2006.

[29] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 357–371, 2018.

[30] Brett D Higgins, Azarias Reda, Timur Alperovich, Jason Flinn, Thomas J Giuli, Brian Noble, and David Watson. Intentional networking: Opportunistic exploitation of mobile network diversity. In *Proceedings of the 16th Annual International Conference on Mobile computing and networking (MobiCom)*, pages 73–84, 2010.

[31] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017.

[32] Javascript HLS client using Media Source Extension. https://github.com/video-dev/hls.js/. Last accessed April 2021.

[33] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 95–110, 2018.

[34] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 187–198, 2014.

[35] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[36] Minsung Jang, Hyunjong Lee, Karsten Schwan, and Ketan Bhardwaj. SOUL: an edge-cloud system for mobile applications in a sensor-rich world. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing (SEC)*, pages 155–167, 2016.

[37] Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A. Chou, Venkat Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, and Hui Zhang. VIA: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[38] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with FESTIVE. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT)*, pages 97–108, 2012.

[39] Matt Klein. Lyft's Envoy: Experiences operating a large service mesh. USENIX SREcon Americas, March 2017.

[40] Kubeedge. https://kubeedge.io/en/. Last accessed September 2021.

[41] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 151–165, 2015.

[42] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[43] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 467–483, 2016.

[44] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, page 122–1225, 2019.

[45] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *Proceedings of the 13th European conference on computer vision (ECCV)*, pages 740–755. Springer, 2014.

[46] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 693–708, 2017.

[47] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 197–210, 2017.

[48] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 197–210, 2017.

[49] The 5 biggest cloud computing trends in 2021. https://www.forbes.com/sites/bernardmarr/2020/11/02/the-5-biggest-cloud-computing-trends-in-2021/. Last accessed September 2021.

[50] Norman Maurer and Marvin Allen Wolfthal. *Netty in Action*. Manning Publications, 1st edition, 2015.

[51] Dirk Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[52] Microsoft's March 3 Azure East US outage: What went wrong (or right)? https://www.zdnet.com/article/microsofts-march-3-azure-east-us-outage-what-went-wrong-or-right/. Last accessed May 2021.

[53] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[54] Brian Noble, M. Satyanarayanan, Dushyanth Narayanan, Eric Tilton, Jason Flinn, and Kevin Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, 1997.

[55] Shadi Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge-computing. *GetMobile: Mobile Computing and Communications*, 24, 2020.

[56] Yanyuan Qin, Shuai Hao, Krishna R Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 366–378, 2018.

[57] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govidan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys)*, June 2011.

[58] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. Online dynamic reordering for interactive data processing. In *Proceedings of the 25th international conference on Very Large Data Bases Conferences (VLDB)*, pages 709–720, 1999.

[59] Mark D Robinson, Ashley R Branham, Andrea Locklear, Sandy Robertson, and Tonda Gridley. Measuring satisfaction and usability of FaceTime for virtual visits in patients with uncontrolled diabetes. *Telemedicine and e-Health*, 22(2):138–143, 2016.

[60] Rahul Sharma and Avinash Singh. *Getting Started with Istio Service Mesh: Manage Microserves in Kubernetes*. Apress Springer, 2019.

[61] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bevier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proeceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 275–287, March 2007.

[62] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. BOLA: Near-optimal bitrate adaptation for online videos. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9, 2016.

[63] Nik Sultana, Salvator Galea, David Greaves, Marcin Wójcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, et al. Emu: Rapid prototyping of networking services. In *Proceedings of the 2017 USENIX Conference on Annual Technical Conference (ATC)*, pages 459–471, 2017.

[64] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very Large Data Bases Conferences (VLDB)*, pages 309–320, 2003.

[65] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very Large Data Bases Conferences (VLDB)*, pages 159–170, 2007.

[66] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd international conference on Very Large Data Bases Conferences (VLDB)*, pages 799–810, 2006.

[67] David L Tennenhouse, Jonathan M Smith, W David Sincoskie, David J Wetherall, and Gary J Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[68] Testing DASH and HLS streams on Linux. http://ronallo.com/blog/testing-dash-and-hls-streams-on-linux/. Last accessed May 2021.

[69] Introducing AthenaX, Uber engineering's open source streaming analytics platform. https://eng.uber.com/athenax/. Last accessed March 2021.

[70] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 374–389, 2017.

[71] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015.

[72] Jue Wang and Michael F Cohen. Very low frame-rate video streaming for face-to-face teleconference. In *Proceedings of IEEE Data Compression Conference*, pages 309–318, 2005.

[73] What is eBPF? https://ebpf.io/what-is-ebpf. Last accessed September 2021.

[74] What is Envoy. https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy. Last accessed September 2021.

[75] wrk2. https://github.com/giltene/wrk2. Last accessed May 2020.

[76] Ying Xing, Stan Zdonik, and J-H Hwang. Dynamic load distribution in the Borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 791–802. IEEE, 2005.

[77] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[78] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment*, 12(5):516–530, 2019.

[79] Xinggong Zhang, Yang Xu, Hao Hu, Yong Liu, Zongming Guo, and Yao Wang. Profiling Skype video calls: Rate control and video quality. In *Proceedings of The 31st Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2012.