Functional Control: Leveraging Function-as-a-Service Platforms for Software-Defined Networking Controllers

Shuwen Liu
sliu9@wpi.edu
Worcester Polytechnic Institute
Worcester, MA, USA

Craig A. Shue
cshue@wpi.edu
Worcester Polytechnic Institute
Worcester, MA, USA

ABSTRACT

The function-as-a-service (FaaS) paradigm, often called "serverless computing," allows computing providers to scalably perform short-lived computational tasks at low cost. While the benefits of FaaS have been demonstrated for ephemeral computational tasks, the research community has not fully explored the applicability of FaaS platforms for longer-term, periodic tasks, like network controllers.

In this work, we explore the fusion of FaaS platforms with the controllers used in the software-defined networking (SDN) paradigm. Traditional SDN controllers are conceived as long-lived, logically-centralized, critical infrastructure for the networks they control. While such "always on" functionality would be a poor fit for the FaaS model, we re-envision the SDN controller as a small, distributed, and tailored entity that handles a single `PACKET_IN` request per instance. We explore this approach with edge computing environments from content delivery network (CDN) providers. In a CDN deployment, we observed a median `PACKET_IN` response time of 15.8 ms from a residential host. The controllers can scalably support both stateful and stateless policy using provider databases and can support geographically-distributed organizations while providing logically-centralized management.

CCS CONCEPTS

• **Networks** → **Cloud computing**; **Network management**.

KEYWORDS

Cloud Computing, Software-Defined Networking, Edge Computing

ACM Reference Format:
Shuwen Liu and Craig A. Shue. 2025. Functional Control: Leveraging Function-as-a-Service Platforms for Software-Defined Networking Controllers. In *International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc '25), October 27–30, 2025, Houston, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3704413.3764448

1 INTRODUCTION

The function-as-a-service (FaaS) paradigm, which is commonly called "serverless computing" since FaaS applications do not require dedicated servers, allows software developers to write simple functions using a computing provider's API to perform distributed tasks. These self-contained functions run on the provider's compute nodes. The provider may distribute these functions arbitrarily and independently across the servers they own, without need for server affinity. This approach can increase programming productivity and improve scalability while decreasing management costs [24]. These platforms often provide distributed databases for storing records.

While FaaS platforms have proven value for short-lived operations, the research community has not fully explored FaaS support for more traditionally long-running jobs. A software-defined networking (SDN) controller is a useful example of a long-running process. The controller's identity is well-known and often manually and statically configured in the SDN agents on the switches that consult that controller. While controllers may be implemented in a redundant or distributed fashion, these controllers are largely viewed as logically-centralized and closely synchronized. These controllers need to have low latency, distributed processing, and scalability [2] and may be deployed at the network edge [1].

Adopting a serverless architecture for SDN controllers offers scalability, cost efficiency, and reduced latency. FaaS platforms offering automatic scaling allow an SDN controller to handle sudden spikes in control plane load without the need for pre-provisioning. For deployers, the pay-per-use model eliminates idle-time costs and infrastructure maintenance costs, since those are handled by the FaaS providers. Many FaaS platforms can automatically use nodes that are close to clients for executing functions, lowering network latency. The FaaS service platforms provide always-on support to functions, ensuring an SDN controller instance can respond to a client's request without maintaining persistent servers. All these properties make FaaS a compelling option for SDN controllers. While adopting a serverless architecture may introduce additional complexity to the SDN control plane, the deployment complexity is minimal, since the FaaS platform manages the function logistics.

To gain all the benefits of the FaaS model, the FaaS provider's platform must be able to dynamically instantiate all parts of an SDN controller that only exists while processing and responding to a given SDN flow rule request. In particular, the design must be always-available to the clients and avoid any centralized components that could become a single point-of-failure. A FaaS design eliminates computational overheads in between SDN agent requests, supports scaling via parallelization, and allows the platform provider to arbitrarily select computational nodes for handling a given request. The FaaS model naturally supports scalable operations since it spawns a function instance automatically to serve each request. A FaaS SDN controller can thus gain the aggregate capacity of the available computing power of the cloud provider's platform. Finally, for geographically distributed organizations, the FaaS model may dramatically reduce SDN agent-to-controller latency when the FaaS platform is able to service agent requests on

MobiHoc '25, October 27–30, 2025, Houston, TX, USA

ACM ISBN 979-8-4007-1353-8/25/10
https://doi.org/10.1145/3704413.3764448

nearby nodes. Prior work [37] found that cloud-hosted SDN controllers could provide reasonable support even for widely-varying residential networks using only public cloud infrastructure. With support for FaaS languages like WebAssembly [41] at content delivery network (CDN) nodes [18], providers may leverage existing edge infrastructure to minimize network propagation times.

While the research community has made an initial foray into exploring FaaS SDN controllers, prior work has not fully implemented a FaaS controller that provides all the FaaS model benefits. Banaie and Djemame [6] created a long-running, centralized platform using the ONOS [33] SDN controller. While the ONOS controller uses OpenFaaS [17] to run individual controller application functions, the ONOS base itself remains a centralized always-on component that could be a single point of failure. FaaS platforms improve fault tolerance through distributed, stateless execution and built-in retries, reducing single points of failure [35]. Therefore, we propose an SDN controller deployment model that avoids any centralized components that could be such single points of failure so that the entire control plane benefits from the automatic scalability and resilience to availability attacks that FaaS platforms enable.

The deployment scenario for a network can greatly affect the rate at which an SDN controller receives requests from SDN agents. In a fully proactive rule deployment, in which the SDN controller specifies in advance all the rules that should be cached by an SDN agent, the controller may only be consulted rarely (e.g., when a switch boots and needs a copy of the rules). In a reactive deployment model, the controller may be consulted more frequently, such as when new flows are created or a cached entry expires. In a host-based SDN deployment model [13], the controller may communicate directly with endpoints and only receive queries when the endpoint is active and establishing new connections. In the case of a host-based mobile device SDN agent that provides per-connection access control [28], the SDN agent may issue multiple requests to a controller in a short window, with long periods of inactivity. Our target use cases include edge deployments where SDN controller workloads are inherently distributed and bursty, such as in host-based SDNs or multi-site organizational networks. A serverless deployment model is particularly attractive since it provides cost-efficient scaling and automatically executes functions on the platform nodes closest to the demand without the need for planning deployment locations. The serverless controller design is also compatible with P4-based data planes [8] that perform actions, such as rule installation or telemetry collection, based on instructions from the FaaS SDN controller.

With this context, we explore five research questions (RQs):

RQ1. To what extent can an SDN controller leverage the FaaS computing paradigm's efficiency, scalability, and performance benefits?
RQ2. To what extent can an SDN controller work with distributed and semi-centralized FaaS computing platforms?
RQ3. To what extent can a FaaS SDN controller adapt to residential networks, organizational networks, and IoT networks?
RQ4. To what extent can a FaaS SDN controller implement various policy sets, including stateful policy?
RQ5. What end-to-end experience would a FaaS SDN controller offer to geographically-distributed clients?

In exploring these questions, we contribute the following:

Create a WebAssembly SDN controller and Publish it on FaaS Computing Platforms: We create an SDN controller in the Rust programming language and then we compile it into a WebAssembly application for the WasmEdge [40] runtime (Section 3). We deploy this SDN controller on the Fastly Compute platform [18] with distributed computing nodes and the Amazon Web Service (AWS) Lambda platform [3] nodes.

Compare Server-based and FaaS SDN Controllers: We compare the prototype's performance with popular open-sourced controllers (Section 4.1). We evaluate each with four metrics: the local response time, the cold start start-up latency, the end-to-end round-trip time, and the usage of computational resources (e.g., CPU and RAM).

Evaluate the Impact of CDN and Various Types of Networks on FaaS SDN Controllers: We deploy the SDN controller on two popular serverless computing platforms. We compare SDN agent request response times from a controller hosted at our organization, in a cloud data center, and on CDN nodes (Section 4.2). We explore support for IoT, residential, and corporate networks (Section 4.3), the support for stateful and stateless policy (Section 4.4), and the impact of varying SDN agent locations (Section 4.5). Our results show CDN-hosted controllers can quickly respond to SDN agents while scaling to meet high demand.

2 BACKGROUND AND RELATED WORK

We discuss the cloud computing model and its supporting techniques and existing SDN deployments.

2.1 FaaS, WebAssembly, and Edge Computing

The cloud computing model has been popular with industry with varying deployment models [24], which range from Platform-as-a-Service (PaaS), that allow full-fledged virtual machines, to the Function-as-a-Service (FaaS) model, which is associated with microservices. The FaaS serverless computing approach offers scaling advantages and opportunities to reduce cost by deploying microservices [21, 39]. Boucher et al. [9] provide an example of an access-control proxy that uses a recurring microservice pattern in which the proxy receives an API request from a user, validates it, and accesses a backend storage service.

Djemame [16] describes technologies and challenges in serverless computing. They summarize several underlying metrics considered in the serverless applications: communication performance, start-up latency, stateless overhead, and resource efficiency. They argue that care must be used in the composition of serverless applications to avoid high communication overheads, particularly when serverless functions are composed in a chain. McGrath and Brenner [30] note serverless computing offers scaling advantages, but introduces challenges for coherent function management.

WebAssembly [41] is a binary instruction format designed as a portable compilation target, enabling deployment on the web for client and server applications. The growth of WebAssembly offers promise for serverless computing. Gackstatter et al. [20] introduce a prototype WebAssembly runtime for Apache OpenWhisk. They show their prototype reduces cold-start latency by up

to 99.5% compared to the standard container-based runtime for various serverless workloads. Kjorveziroski and Filiposka [25] evaluate the execution performance of WebAssembly runtimes integrated with Wasmtime and find that the WebAssembly runtimes achieve better execution times than directly running in a container, but required more computation time for intensive operations.

WebAssembly applications in a serverless platform can support network interactions via an HTTP API. Lee et al. [26] evaluate the concurrent invocation of different trigger types to demonstrate the performance and throughput of serverless computing. Their experiments shows HTTP triggers have reasonable concurrent request processing support. With overlay HTTP messages, an SDN client can query a controller request messages.

Edge computing aims to place services as close to endpoint devices as possible. For example, Cicconetti et al. [14] introduce a framework for IoT devices that includes serverless edge computing. They suggest that every edge node (or small cluster of edge nodes) could host a serverless platform, such as OpenWhisk [4].

2.2 SDN Deployments

SDN divides the traditional network routing platform into a control plane and a data plane. The controller is required to communicate with the data plane for managing network applications. Aditya et al. [1] describe needs for SDN adoption, including resources for virtualized network function execution. They also note many network applications are short-lived and cannot afford high start-up latency. This places pressure on SDN controllers to quickly serve each incoming request. They note that the serverless computing paradigm could be used for implementing SDN controllers.

Comer and Rastegarnia [15] introduce a distributed microservice architecture that divides the monolithic SDN controller into a set of cooperative microservices to avoid overheads. Banaie et al. [6] introduce a serverless computing platform with a layer for the SDN core service, a communication interface, and a set of serverless functions. The core service is a long-term ONOS open-source SDN controller. The core service transmits events to a specific FaaS function through the communication interface. This "always-on" core design consumes computing resources for the core service without fully leveraging the FaaS model.

Our approach fuses the FaaS model with SDN controllers using WebAssembly and distributed compute nodes with the aim to offer high performance and low-latency to end devices while capitalizing on the resource efficiency and scalability associated with the FaaS model. We avoid long-running SDN components and instead use providers' triggers to spawn SDN controller instances.

3 A WEBASSEMBLY SDN CONTROLLER

We describe how we transform the traditional long-running SDN controller to deployments of serverless computing. Our implementation's source code is publicly available [27].

The Rust programming language can be used to create standalone executables and to compile into other languages, such as WebAssembly. To compare and contrast a FaaS controller with a server-based one, we start by creating a Rust-based SDN controller. Our implementation uses the `rust-ofp` [7] crate to implement an OpenFlow protocol-compatible SDN controller.

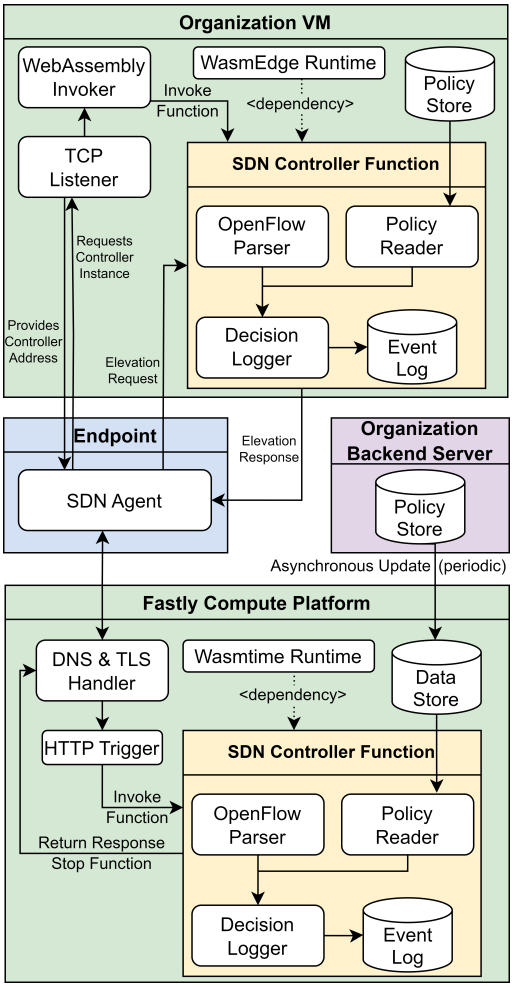

Figure 1: FaaS SDN controllers can be deployed on local infrastructure (top green box) or on FaaS providers' platforms (bottom green box).

In Figure 1, we provide a diagram of our WebAssembly SDN controller prototype. In the upper portion of the diagram, we show how an organization could have a minimal long-running TCP listener that could trigger the instantiation of an SDN controller function for processing. That controller is activated, processes a single OpenFlow request, sends a response, and terminates. To create it, we compile our Rust controller program into a WebAssembly application binary. To support TCP streams, we use the `WasmEdge` Runtime [40] to create a TCP listener process and an invoker process to launch our compiled SDN controller function.

Prior work found that the round-trip time (RTT) between the SDN agent and the SDN controller can have a significant impact on end-user experiences with applications such as web browsing [37] and traffic network control [11]. Since CDNs are designed to position resources closer to client endpoints to reduce RTTs and the number of network links that must transmit packets, they may be candidates for hosting FaaS SDN controllers.

In the lower portion of the diagram, we show how this ephemeral SDN controller can be integrated into a CDN provider's platform.

To provide a concrete example, we illustrate our use of the Fastly Compute Platform; with modest changes, we also use our approach with AWS Lambda.

In the Fastly platform, a developer registers a FaaS application and receives a DNS hostname that clients may use to access the application. When a client issues a request for the host name associated with our SDN controller, Fastly's DNS infrastructure selects the appropriate CDN node, often the node with the lowest latency to the client, to process the request and returns that node's IP address. The client then connects to an HTTP server running on that CDN node. Since each node services multiple customers, the HTTP server uses the requested host name to determine the appropriate customer function to call. The HTTP server completes a TCP handshake with the client, creates an instance of our SDN controller function, and then forwards the HTTP request to that controller instance. As it is launched, the SDN controller function can pull configuration information from a Fastly database. The SDN function processes the request and sends a response to the client via the Fastly HTTP server. The function logs entries to the Fastly database and terminates. While we used the `WasmEdge` Runtime on our own server, we used the `WasmTime` [10] Runtime for deploying to Fastly.

There are significant differences between our approach and that of Banaie et al. [6]. They use the ONOS controller to host a long-term SDN core server that dispatches specific services to serverless function. In contrast, we transform the whole SDN controller into a truly FaaS application. Whether our SDN function is deployed to Fastly's platform or not, the DNS and HTTP infrastructure is in place to service other client requests. When we deploy our SDN controller function on the Fastly infrastructure, the platform does not immediately start any processes associated with our SDN controller. The platform only runs our function to respond to a given client's OpenFlow `PACKET_IN` request and the function terminates upon sending a response and logging the event.

4 EMPIRICAL STUDY: FAAS DEPLOYMENT

Our earlier research questions were: *To what extent can an SDN controller leverage the FaaS computing paradigm's efficiency, scalability, and performance benefits? To what extent can an SDN controller work with distributed and semi-centralized FaaS computing platforms? To what extent can a FaaS SDN controller adapt to residential networks, organizational networks, and IoT networks? To what extent can a FaaS SDN controller implement various policy sets, including stateful policy? What end-to-end experience would a FaaS SDN controller offer to geographically-distributed clients?*

In Table 1, we show how we explore these research questions across five phases. We vary a single parameter in each phase. We select and hold constant the bold face option in each phase's list for all the other phases. In Phase 5, we additionally look at a FaaS controller variant that emulates the OpenFlow handshaking process. In all other phases, we use a process where the `HELLO`, `SWITCH_FEATURES`, and `PACKET_IN` messages are combined into a single request. This simplified message set is used as a proof-of-concept to evaluate the feasibility and performance of our serverless controller design. While this abstraction does not fully implement the OpenFlow specification, it enables an evaluation of the performance properties of SDN applications that require a controller's vetting of each new flow.

In Phase 1, we explore the impact of programming language and design structure on controller performance. We construct a baseline of a set of traditional, open-source server-based SDN controllers. We then examine our prototype SDN controller, both as a more traditional server-based Rust implementation and with our FaaS WebAssembly variant. We use the `cbench` [31] and `perf` [34] benchmarking tools to compare these controllers across a variety of metrics to capture efficiency, scalability, and performance characteristics. We deploy each controller in a VM at our organization with a straightforward IP address blocklist policy. For all subsequent phases, we use our WebAssembly controller implementation.

In Phase 2, we deploy the WebAssembly controller on two popular FaaS computing platforms and our organization server. We share the code for our implementation in a GitHub repository [27]. These controllers are hosted by distributed, semi-centralized, and centralized computing nodes, letting us observe the impact of controller locations and distribution. In Phase 3, we explore prior data sets of network traffic in an IoT network, a residential network, and an corporate/industrial network. By replaying that traffic, we can see how a FaaS controller would perform. In Phase 4, we explore varied policy sets. While packet header rule policies are common in controller implementations [32, 33], we also include a policy based on a state machine to explore the extent to which stateful policies can be supported. In Phase 5, we vary the SDN agent location to observe the end-to-end performance.

4.1 Efficiency, Scalability, and Performance

We explore efficiency, scalability, and performance of the implementations across a series of experimental scenarios, starting with the processing time and query throughput rate.

The amount of time it takes for a controller to process a request is a critical measure of the controller's performance. We consider the response time to be the time between when a client sends the controller a `PACKET_IN` request and when that client receives the controller's `PACKET_OUT` response. To eliminate network propagation factors, we use a client hosted on the same system as the controller. We examine the response time of four SDN controllers including: POX (gar version), Floodlight v1.2, our SDN controller running in Rust, and our WebAssembly SDN controller (using the `WasmEdge` runtime [40]). The POX controller [29] is written in Python and the Floodlight controller [19] is written in Java.

We install each controllers on a virtual machine with two cores and 8 GBytes of memory. We use the `cbench` benchmarking tool [31] which is commonly used to profile SDN controllers and has microsecond (μs) timing resolution. During the experiments, we configure `cbench` to send a `PACKET_IN` and wait for a `PACKET_OUT` to return. We then count the number of sequential `PACKET_OUT` responses received in one second. We calculate the mean response time by dividing one second by the number of `PACKET_OUT` responses received. Each second represents one mean response time data point; we collect 1,000 such data points for each controller.

In Table 2, we show results of the mean response times across 1,000 trials for each of the four controllers. The low standard deviation shows that the results across trials were fairly consistent.

Table 1: To feasibly explore multiple parameters related to serverless SDNs, we vary only one parameter in each phase and hold that parameter steady in other phases. This table summarizes the explored parameters in Section 4.

Phase and Research Questions Explored	Phase 1: Controller Implementation Impact	Phase 2: Controller Location Impact	Phase 3: Type of Underlying Network	Phase 4: Types of Policy Implemented	Phase 5: Client Location Impact
Options for the parameter that is varied in the indicated phase. The **bold face** option is used for that parameter in all other phases.	• Floodlight • Pox • Rust • **WebAssembly**	• **Fastly Compute (Distributed)** • AWS Lambda (Semi-Centralized) • Centralized Server at our Organization	• IoT SDN • Residential SDN • **Organization SDN**	• IP Address Blocklist • **Flow Header Rules** • State Machine Rules	• **Residential Network Computer** • VM at Our Organization • AWS EC2 Instance • Alibaba Cloud VM

The Floodlight controller has the lowest mean response time; the median of its data points is 28.4 microseconds (μs). The Rust and WebAssembly variants of our controller have response times that are close to each other, with median response times of 55.0 μs and 52.4 μs, respectively. This similarity is likely due to being compiled from the same source code. The POX controller shows a higher response time than other three controllers with a median result of 91.0 μs. In this table, and in many of the following tables, the standard deviation is affected by data points that are significantly higher than the rest of the distribution. These outliers could be due to unrelated activity in the network or system. We simply note their presence and preserve them in the reported data.

Table 2: The results of mean response time of tested controllers. Each series includes 1,000 independent second-long trials of the controller response rate.

	Mean Response Time (microseconds)			
Controller	10th percentile	median	90th percentile	std. dev.
POX	72.1	91.0	93.0	7.0
Floodlight	26.6	28.4	30.1	5.2
WebAssembly	50.8	52.4	54.4	1.6
Rust	49.2	55.0	58.5	4.1

We then use `cbench` to examine the maximum throughout rate that each controller can achieve under the same test environment. We configure `cbench` to continuously send and queue `PACKET_IN` requests while the buffer is not full. We count the number of sequential `PACKET_OUT` responses received in one second and calculate the throughput rate as the number of responses received in one millisecond. Each second represents one throughput rate data point and we collect 1,000 data points for each controller.

Table 3 represents the results of the throughput rates across 1,000 trials for each tested controller. The Rust controller has the highest throughput rate as the median of its data points is 124.5 packets per millisecond (ms). The WebAssembly controller achieves a slightly lower throughput rate compared to the Floodlight controller, with median throughput rate of 65.3 packets per ms and 71.9 packets per ms, respectively. The POX controller shows the lowest throughput rate with a median result of 40.6 packets per ms.

While query response time and throughput rate are important, the resource usage of the controller implementations affects the deployment costs. When we examine the resources used by the four SDN controllers, we first observe their usage of CPU and memory after completing the initial start-up (e.g., binding the OpenFlow port and awaiting the first client's connection). We show these results in the second and third columns of Table 4. We note that the Floodlight controller consumes around 5% more CPU and 280 Mbytes more than the other controllers. The Rust controller consumes the least CPU and memory among the tested controllers.

Table 3: The results of throughput rate of tested controllers. Each series includes 1,000 independent second-long trials of the controller throughput rate.

	Throughput Rate (packets per millisecond)			
Controller	10th percentile	median	90th percentile	std. dev.
POX	36.1	40.6	42.8	2.9
Floodlight	66.0	71.9	77.3	7.3
WebAssembly	61.5	65.3	74.9	5.0
Rust	108.4	124.5	133.0	10.1

Table 4: CPU and RAM usage of the tested controllers

	Avg. Idle Usage		Avg. Active Usage	
Controller	CPU (%)	RAM (MBytes)	CPU (%)	RAM (MBytes)
POX	0.3	14.7	41.1	14.8
Floodlight	5.2	296.8	56.2	529.2
WebAssembly	0.3	13.1	14.3	13.1
Rust	0.1	0.8	20.8	79.0

We then collect the resource usage while these controllers are active and handling `PACKET_IN` messages sent from another virtual machine with a packet arriving rate of 1,000 packets per second. Each of the controllers have increased CPU usage, with the Floodlight controller still consuming the most CPU cycles. While the memory usage of the WebAssembly and POX controllers are largely stable, the Floodlight and Rust controllers consume significantly more memory than during the idle stage. The WebAssembly controller, which is built with FaaS deployment in mind, has good resource usage during high-demand bursts.

We examine the number of CPU cycles used by these controllers using the `perf` tool [34] in two measurements. The first measurement reports the cycles used to start and then immediately terminate the SDN controller. The second measurement records CPU cycles used during one second after the controller has started. In Table 5, we see results that are consistent with CPU usage percentages. The WebAssembly controller consumes the least CPU cycles to simply start and exit while the Floodlight controller consumes the most CPU cycles. The `perf` tool did not report any CPU cycles used by the WebAssembly or Rust controllers during the idle second. Between the Floodlight controller and the POX controller, Floodlight has significantly higher CPU cycle usage in the idle second period.

Table 5: CPU cycles used by tested controllers for start-exit and idle

	Avg. CPU cycles used in	
Controller	start-exit	idle for one second
POX	449,907,900	344,584
Floodlight	7,496,986,849	2,075,673
WebAssembly	38,826,770	[none reported]
Rust	146,909,126	[none reported]

Finally, we examine the amount of time required to initiate each of the SDN controllers and have them ready to serving new clients. While this factor is less important for long-running SDN controllers, since the servers infrequently restart, it is important for controllers that regularly start and stop, such as in the FaaS model. In the same virtual machine that runs previous experiments, we develop a timing program to record two timestamps. The first timestamp (t_1) is the time at which the program invokes the SDN controller. Immediately upon invoking the controller, our measurement program continuously sends packets to attempt to connect to the controller. The second timestamp (t_2) represents the time at which the SDN controller successfully replies to a connection request. We consider the difference ($t_2 - t_1$) as the start-up latency of the controller.

Table 6: We compared the start-up (cold start) latency of four controllers. Each series includes 1,000 independent trials of the controller start-up.

	Start-up Latency (milliseconds)			
Controller	10th percentile	median	90th percentile	std. dev.
POX	263.8	283.2	306.5	17.8
Floodlight	4,619.6	4,735.8	4,843.0	250.3
WebAssembly	11.6	12.1	16.7	2.0
Rust	70.0	78.8	85.1	6.4

In Table 6, we show the start-up latency of the four controllers across 1,000 trials for each controller. The Floodlight controller spends a much longer time in its starting process as compared to other controllers; it takes a median of 4,735.8 ms for Floodlight to be ready to accept its first client. The POX and Rust controller have median start-up latencies of 283.2 ms and 78.8 ms, respectively. The WebAssembly controller has the shortest start-up latency among these controllers, with a median of 12.1 ms. This highlights the `WasmEdge` runtime's success at minimizing the start-up cost of FaaS applications.

In summary, the results we present in this section show there are tradeoffs inherent in SDN controller designs. The Floodlight controller has median response time that is just over half that of the next closest competitor, but it pays for that in CPU and RAM usage and a longer start-up time. The WebAssembly controller starts quickly and has, at worst, the second-lowest CPU usage and memory usage. These tradeoffs align with the long-running deployment design of Floodlight and the shorter-lived design for WebAssembly.

4.1.1 Concurrency and Scalability. In the FaaS model, the provider platform automatically spawns SDN controller instances on demand. The platform provider may perform load balancing and choose which node to host each new instance. For CDN platforms, the selection often prioritizes CDN nodes with lower latency to the client. This CDN mechanism has been effective at handling large "flash crowds," with billions of requests during spiking demand [42].

We explore the impact of running four concurrent SDN agents elevating traffic to the FaaS SDN controller on the Fastly CDN platform. We use the organization network data set [23], which we describe in more detail in Section 4.3. To avoid exact synchronization between SDN agents, we replay four distinct business hours of the organization network traffic from Tuesday to Thursday in each of the four agents. We host these SDN agents in different geographic regions and each agent transmits a `PACKET_IN`request for each new TCP flow it witnesses. These SDN agents record the elapsed time between sending a request and receiving a response.

In Table 7, we show the results of the experiments. Given that distributed nodes are generally able to accurately synchronize clocks to the nearest second, we use a second as the time resolution for the study. We mark an SDN controller instance as running for each second in which it receives a request. With our earlier throughput rate findings of tens of requests per millisecond and results in Section 4.2 showing full SDN request round-trip times in the tens of milliseconds, the second granularity we use may overestimate the degree of concurrency since multiple requests could occur serially in the same one-second duration window. Nonetheless, we see in Table 7 that over 25% of the 3,600 seconds in the hour-long experiment had zero controller instances running and another 37% had only a single controller instance running. With an organizational data set in which the SDN agent issues a controller review request for every new flow, at most 37% of the time had concurrent requests. Of the 1,329 seconds with concurrent requests, over 71% (960) had only two concurrent SDN controller instances.

The degree of concurrency and the number of requests per hour of a real-world organization appear low in comparison to the capacity required for CDN platforms to support flash crowds.

4.2 Impact of Distributed Deployment

The round trip time (RTT) between the client application and the SDN controller makes a significant impact on users' experiences [37]. When considering a residential network, does a CDN provider have a lower latency than a near-by server? We empirically explore the RTT between the SDN agent's request and its receipt of the controller's response.

Table 7: Concurrency measurement when four geographically-distributed SDN agents issue per-connection controller review requests based on data from the organization data set on of one-hour periods selected from different week days.

	Response Time (milliseconds)			# of	Number of Seconds in which k Controller Instances Run				
Client Location	10th	median	90th	Packets	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
VM at Our Organization	12.3	13.0	13.8	85,755					
AWS EC2 Instance in US West	3.3	3.9	4.6	35,905	924	1,347	950	321	58
AWS EC2 Instance in US East	2.3	3.1	4.0	27,472	(25.7%)	(37.4%)	(26.4%)	(8.9%)	(1.6%)
Alibaba Cloud VM in Japan	4.8	5.3	5.7	38,209					

We first install the Fastly Command-Line Interface (CLI) and the Fastly Compute platform [18] on a server in our organization. The Fastly CLI allows us to use our own infrastructure with the Fastly Compute software, including a local toolchain with features for creating, debugging, and deploying WebAssembly services. This lets us "deploy" the WebAssembly SDN controller on an instance running on our own organization's server as a baseline (using our own organization's DNS host names). We then publish the WebAssembly SDN controller on the Compute platform built on Fastly's CDN nodes. Fastly automatically generates a domain name for the service, which can be accessed by the test client to access the CDN node instances.

Then we install a Rust runtime [5] for AWS Lambda [3] on the same server in our organization. We use the Rust runtime client to build a Lambda function of the SDN controller locally. Lambda runs the compiled executable directly and does not require a WebAssembly executable, unlike Fastly. As with Fastly, we can use this local controller instance with our test client. We likewise publicly deploy the function of the SDN controller on AWS Lambda services that is served by an AWS server in the US East data center. The test client uses an AWS Lambda host name to access the service.

To evaluate the performance of the FaaS SDN controllers deployed on different platforms and servers, we use an SDN agent running on a laptop in a residential network. From the client, we repeatedly send an HTTP request wrapping the OpenFlow `PACKET_IN` message to the domains of four tested controllers, wait for the matched response, and record the elapsed time.

Table 8: The SDN agent request response time for SDN controllers running the Fastly and AWS Lambda platform software, both running on our server (self-hosted) and on the platforms' respective servers. Each row has data from 34,974 independent trials.

	Response Time (milliseconds)			
Platform	10th	median	90th	std. dev.
Compute (Distributed)	14.1	15.8	20.2	7.4
Compute (Self-Hosted)	22.5	24.5	27.3	3.1
Lambda (Semi-Centralized)	37.0	41.8	48.9	14.8
Lambda (Self-Hosted)	25.1	26.2	29.2	3.0

In Table 8, we show the results of data points of response time collected from the above experiments. By replaying one hour of traffic from an organization's network [23], we use the same SDN agent to conduct 37,974 trials to each of the four different SDN controllers: a) the controller running on Fastly's CDN nodes, b) the Fastly SDN controller instance running on our organization's server, c) the AWS Lambda instance running in the AWS US East data center, and d) the AWS Lambda instance running on our organization's server. The two self-hosted results are similar; while these show Fastly's software stack has slightly better performance characteristics for our prototype than the AWS Lambda instance, future optimization efforts may eliminate that difference.

In exploring the results when the platforms host the instances themselves, we see that the Fastly CDN has significantly lower request response times than AWS Lambda. This difference is likely due to the server deployment models of the providers: Fastly's highly distributed CDN aims to place servers geographically close to clients while AWS Lambda uses a semi-centralized approach using a smaller number of larger data centers that serve larger geographic regions. With network propagation delays potentially being a significant component of the differences, it is possible the Fastly server is simply closer to the client.

The RTT between the SDN agent and the SDN controller can affect the user experience. Prior work found using a cloud-hosted controller at a 50 ms RTT would add a two-second delay to the web page load time in the worse case scenario [37]. With lower RTTs associated with distributed controllers, the page load time for clients would decrease, improving the user experience. By using distributed CDN infrastructure, deployers may minimize RTTs for many SDN agents.

4.3 Home, IoT, and Organization Networks

With the potential for highly distributed FaaS SDN controllers, we explore the research question: *To what extent can a FaaS SDN controller adapt to residential networks, organizational networks, and IoT networks?* We leverage prior data sets from varied networks including home networks, IoT device networks, and organizational networks. By replaying these network traffic, we evaluate the applicability of the FaaS SDN controller for real-world networks.

We use public network data sets that were generously provided to the research community. Vaccari et al. [38] create an IoT network composed of eight IoT sensors and record the network traffic for seven days (which we refer to as the "IoT Network"). Hjelmvik [22] performs network forensic analysis in a real Internet-connected network, which provides 40 days of network traffic (which we refer to as the "Home Network"), including web browsing, chat, and email. The ICS Lab [23] hosts an organizational network composed of various equipment including PLCs, servers, switches, and firewalls. They share three days of captured network traffic from the organization's network (which we refer to as the "Organization Network"). All of these data sets are available in the `pcap` data format.

We create a tool to replay the network traffic and send associated OpenFlow SDN agent messages to the FaaS SDN controller.

By using two Rust libraries, `pcap` and `pdu`, we create a tool to parse raw data sets into packet level information. We collect the flow header information to construct an HTTP request that contains an OpenFlow `PACKET_IN` message as the payload. Since we are replaying traffic, we can choose whether to simply send these HTTP requests as fast as the tool can parse each packet or we can use the timestamp of every packet to match the real-world delay between requests. We provide results from both of these options.

We next consider which packets should be sent to the controller. In some cases, SDN agents may request a controller review for only the first packet in each new network flow, particularly if an SDN controller makes use of `FLOW_MOD` messages to cache decisions at the SDN agent. In other work [12, 28], the SDN agent may request a controller review of all the packets associated with particular network flows to enable stateful policy enforcement at the SDN controllers. We provide results for both deployments, using the "Full Flow Mode" label when the SDN agent sends every packet to the controller and the "Initial Packet Mode" label for SDN agent requests for only the first packet in each flow.

After creating the replay tool supporting two replay modes, we install it on a residential network computer and send requests to the serverless SDN controller deployed on Fastly Compute. We replay packets from a one-day period in the IoT network data source and a similar period for the home network. We use a two-day period for the organization's network data set. For the data sets, we set the packet replaying tool in Full Flow Mode for each and in Initial Packet Mode for the home and organization networks. We do not examine Initial Packet Mode for the IoT Network due to a small number of continuing flows, which yields a small sample size. We measure the performance of the SDN controllers by measuring the time between when the agent issues the HTTP request and when the client receives the HTTP response. Table 9 shows the results of these experiments under the maximum transmission-rate setting. We found most response times range from 12.5ms to 20.8ms.

Table 9: The SDN agent request response times for the FaaS SDN controller and the SDN agents with different request scenarios when maximizing packet arrival rate

	Response Time (milliseconds)			
Network Type	10th	median	90th	std. dev.
IoT Network (Full Flow)	12.5	14.5	19.3	9.0
Home Network (Initial Packet)	14.0	15.6	20.8	6.9
Home Network (Full Flow)	13.2	14.4	17.6	6.8
Organization Network (Initial Packet)	13.4	15.7	20.4	5.1
Organization Network (Full Flow)	13.3	15.2	18.7	6.0

In Table 10, we show the results when the packet replaying tool sends OpenFlow query messages at a pace that matches the real-world arrival rate seen in the packet captures. We replay the traffic from a one-hour period associated with the data sets of the IoT network, home network, and organizational networks. Given the slower arrival rate, we use the Full Flow Mode for each. Table 10 indicates that most response times range from 12.3ms to 19.0ms.

In examining the results from Tables 9 and 10, we find that the FaaS SDN controllers can support the real-time SDN agent requests

Table 10: The response time for the FaaS SDN controller and the SDN agent with different network traffic when replaying real-world packet arrival rate.

	Response Time (milliseconds)			
Network Type	10th	median	90th	std. dev.
IoT Network (Full Flow)	12.3	14.3	17.3	7.9
Home Network (Full Flow)	13.3	15.2	18.9	7.5
Organization Network (Full Flow)	13.1	15.0	19.0	8.4

associated with the IoT, residential, and organizational networks without performance degradation. Even when we expedite the packet captures to transmit at an artificially high rate, the SDN controller results are similar. When hosted on a CDN platform, the SDN controller can scalably support different types of networks.

4.4 Stateless and State-Based Policy

We now explore the research question: *To what extent can a FaaS SDN controller implement various policy sets, including stateful policy?* Even stateless controllers need some form of policy to apply in making decisions. Fortunately, the FaaS computing platform providers supply mechanisms for loading data to be used by the customer's serverless functions.

We use two types of data stores, `ConfigStore` and `KVStore`, provided by the Fastly Compute platform to implement three policy sets. The `ConfigStore` provides storage that is organized in a hash map data structure. Each edge CDN node can read this data structure. That data structure can be updated using a back-end server, which will then be synchronized and propagated globally. This is useful for distributing enforcement policy. For stateless policy, this is the only data structure that we need to read.

For the first set of policy, we compile 500 IP address rules from an open-sourced malicious IP list [36] into a blocklist that we store in the `ConfigStore`. An example policy rule can be to drop a flow if it has source IP or destination IP in the malicious IP list.

As a second set of policy, we explore more detailed matching rules. Prior work [32] indicates that a typical flow rule includes a set of flow match fields, a flow priority, and a flow action set. ONOS [33] supports flow match fields including MAC address, IP address, TCP port, and transport-layer protocol. To replicate such functionality, we create 500 flow rules using a variety of these header fields to compile a set of rules in the `ConfigStore`. As an example, the flow rule policy can support allowing every network flow on port 22.

In other work [12], the SDN controller uses a state machine policy to secure IoT devices in home networks. The state machine policy needs to keep track of the state of every flow. The prior `ConfigStore` data structure cannot support our need to write entries at the CDN node running the controller. Instead, we use Fastly's `KVStore` data structure, which has an "eventual consistency" guarantee (i.e., writes are immediately consistent intra-node and later become consistent between nodes). The `KVStore` allows CDN nodes to both read and write entries. We use this to maintain the flow state when handling each `PACKET_IN`. Specifically, after the platform invokes an instance of the controller function upon receiving a request, such cold-started instance can retrieve its state by reading the `KVStore` from the edge node. This instance also

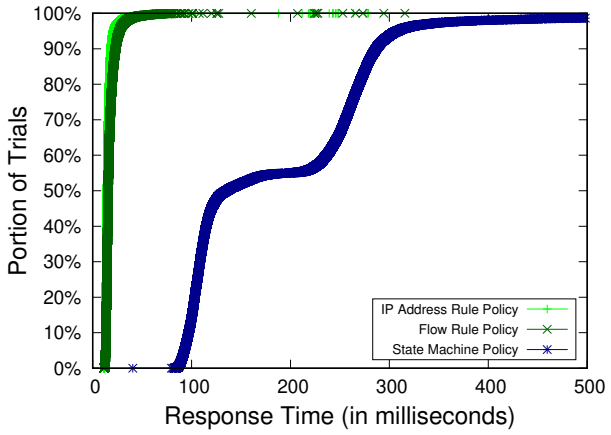

Figure 2: Response times for the experiments for the SDN agent and FaaS controllers with different policy sets. The lines representing the IP address rule policy and the flow rule policy have significant overlaps and appear on the left. The line on the right shows the state machine policy.

reads the state machine rules from the `ConfigStore`. It then sends the response based on its current state and the state machine rules. Finally, it writes the updated state to the `KVStore` which will be available for other edge nodes instantly. We insert our read-only policy of 500 state machine rules into a `ConfigStore`.

We then evaluate the SDN controller's performance when implementing these three policy sets. We use a residential network computer to replay one hour of traffic from the organization network described earlier [23]. We configure the SDN agent to send a request to the controller for every packet in the packet capture to the controller deployed on the Fastly Compute platform and record the time difference between sending the HTTP requests and receiving the HTTP responses.

Figure 2 shows the results of these experiments. We note that the two leftmost lines have significant overlap. They show the data points associated with the IP rule policy and the flow rule policy. The median and 90th percentile data points for the IP rule policy are 14.7 ms and 18.4 ms, which are slightly lower than the median and 90th percentiles of the flow rule policy, which are 15.7 ms and 23.1 ms, respectively. It takes longer for the SDN controller to parse flow rules that include more information than an IP address blocklist.

The results for the state machine policy are quite different: the median and the 90th percentile for the state machine rule policy are 135.6 ms and 286.6 ms, respectively. The approach incurs a significant delay at the SDN controller, likely due to the tracking of the flow's state. This requires frequently read and write operations on the `KVStore` data structure. While the read latency for the `KVStore` is typically under a millisecond, Fastly's documentation [18] notes it may take up to 300ms to read data that has recently changed. The bends in the curve for the stateful policy results align with `KVStore` performance reported in the Fastly documentation.

These results show that stateless policy can be faster than stateful policy, at least in some platforms. Deployers may need to determine whether they can accept such delays for applications that require such stateful analysis, such as the relatively low-traffic IoT applications in prior work. However, since the SDN controllers operate as ephemeral functions that run in parallel, deployers can design their deployments so that stateful and stateless application policies run in separate controller instances. This will allow agents and applications with stateless policy to achieve high performance while only incurring delays for the agents that need stateful policy. In the future, platform providers may choose to optimize data structures like the `KVStore` for lower latency. Until then, deployers may explore other controller deployment models for applications requiring low-latency, stateful policies.

4.5 Geographically Distributed SDN Agents

To examine the impact of the geographic location associated with an SDN agent, we publish the FaaS SDN controller on the CDN nodes associated with the Fastly Compute platform and access it from SDN agents in different networks. We use the DNS host name provided by the Fastly platform for clients to access the service.

We placed a typical SDN agent in four different locations to simulate potential deployment sites. The first SDN agent we explore is installed in a computer in a residential network. A second SDN agent is located in a server at our organization, which may represent a typical corporate network deployment. We then install agents in two public data centers, one in the Amazon Web Services US East data center and one in the Alibaba cloud server in Hong Kong, China. These data center installations can illustrate the impact for well-connected data centers accessing CDN FaaS controllers.

We simulate an organization's usage by replaying the traffic from an organization's network [23] from each of these four SDN agent locations. The SDN agents send review requests for the replayed traffic to the FaaS SDN controller and record the elapsed time between transmitting the SDN agent request and receiving the controller's response. In Table 11, we show the results of an hour of SDN agent requests for the organization network's traffic from clients in these locations, with 34,974 requests from each client. The median request response time is less than 10ms for an SDN agent hosted on our organization server or at either of the data centers. From the residential network, the median response time is 14.1ms.

Table 11: The results of the validation time for clients in different locations with the combined request method

	Request Response Time (milliseconds)			
Client Location	10th	median	90th	std. dev.
VM at Our Organization	9.2	9.7	10.3	6.0
Residential Network Computer	12.9	14.1	17.7	6.1
AWS EC2 Instance in US East	3.1	3.4	4.3	1.7
Alibaba Cloud VM in China	4.7	5.2	6.1	3.1

For all the results we have presented thus far for the FaaS SDN controller, we have used an aggressive bundling of OpenFlow messages. We compiled the OpenFlow `HELLO` message, `PACKET_IN` message, and `SWITCH_FEATURES_REPLY` message into a single HTTP request's payload. This allows the SDN controller to identify clients and handle query messages, but it does not allow for a negotiation between the parties. The trade-off is that it consumes only a single round-trip time for requests rather than three round-trips.

Table 12: The results of the validation time for clients in different locations with the three-way handshake method

	Client Validation Time (milliseconds)			
Client Location	10th	median	90th	std. dev.
VM at Our Organization	18.7	19.5	20.4	5.7
Residential Network Computer	25.7	28.9	37.2	8.6
AWS EC2 Instance in US East	6.5	7.0	7.8	2.0
Alibaba Cloud VM in China	10.4	11.3	13.3	5.1

We explored a faithful recreation of the standard OpenFlow handshake to determine the performance aspects of each. That handshake process includes three steps. The agent sends an OpenFlow `HELLO` to the controller. The controller responds with a `HELLO` and a `SWITCH_FEATURES_REQUEST` message. The SDN agent replies with a `SWITCH_FEATURES_REPLY` message. We record the time when the SDN agent sends its `HELLO` and the time at which the SDN agent receives the controller's acknowledgment of the agent's `SWITCH_FEATURES_REPLY` message. We report the elapsed time between these measurement points across 35,000 trials of the handshake process for each agent with the distributed SDN controller. We show the results in Table 12. The handshake time for all the agents are consistent with low standard deviations. For 90% of trials of tested agents, the controller handshake time is 37.2 ms or less.

5 CONCLUSION

In this work, we explore the viability of a FaaS SDN controller. We find that we can reasonably construct such a controller using WebAssembly and deploy it on commercially-available CDN provider networks. The on-going resource costs are lower than production open source SDN controllers. While the Floodlight controller can respond to requests around 25 μs faster than our WebAssembly implementation, the network propagation times of messages could quickly dwarf that difference. With a cloud provider's natural ability to create multiple instances of the WebAssembly controller dynamically, the approach can easily scale to support even large, busy networks. Finally, leveraging widely distributed CDN nodes can significantly decrease the request latency to SDN controllers, minimizing an impact to the user experience.

REFERENCES

[1] Paarijaat Aditya, Istemi Ekin Akkus, Andre Beck, Ruichuan Chen, Volker Hilt, Ivica Rimac, Klaus Satzke, and Manuel Stein. Will serverless computing revolutionize NFV? *Proceedings of the IEEE*, 107(4):667–678, 2019.

[2] Suhail Ahmad and Ajaz Hussain Mir. Scalability, consistency, reliability and security in sdn controllers: a survey of diverse sdn controllers. *Journal of Network and Systems Management*, 29:1–59, 2021.

[3] Amazon Web Services. Building Lambda functions with Rust. https://docs.aws.amazon.com/lambda/latest/dg/lambda-rust.html, 2024.

[4] Apache Software Foundation. Apache openwhisk is a serverless, open source cloud platform. https://openwhisk.apache.org/, 2024.

[5] AWS-Lambda-Rust-Runtime Contributors. A rust runtime for aws lambda. https://github.com/awslabs/aws-lambda-rust-runtime, 2024.

[6] Fatemeh Banaie and Karim Djemame. A serverless computing platform for software defined networks. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 113–123. Springer, 2022.

[7] Sam Baxter. rust-ofp v0.2.1. https://crates.io/crates/rust_ofp, 2024.

[8] 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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014.

[9] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *USENIX Annual Technical Conference*, 2018.

[10] Bytecode Alliance. Wasmtime. https://wasmtime.dev/, 2024.

[11] Yingqing Chen and Christos G Cassandras. Scalable adaptive traffic light control over a traffic network including transit delays. In *International Conference on Intelligent Transportation Systems (ITSC)*, pages 1651–1656. IEEE, 2023.

[12] Zorigtbaatar Chuluundorj, Shuwen Liu, and Craig A Shue. Generating stateful policies for IoT device security with cross-device sensors. In *IEEE Network of the Future (NoF) Conference*, 2022.

[13] Zorigtbaatar Chuluundorj, Curtis R Taylor, Robert J Walls, and Craig A Shue. Can the user help? Leveraging user actions for network profiling. In *International Conference on Software Defined Systems (SDS)*, pages 1–8. IEEE, 2021.

[14] Claudio Cicconetti, Marco Conti, and Andrea Passarella. A decentralized framework for serverless edge computing in the internet of things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2020.

[15] Douglas Comer and Adib Rastegarnia. Toward disaggregating the SDN control plane. *IEEE Communications Magazine*, 57(10):70–75, 2019.

[16] Karim Djemame. Serverless computing: Introduction and research challenges. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 15–23. Springer, 2022.

[17] Alex Ellis. Serverless functions, made simple. https://www.openfaas.com/, 2024.

[18] Fastly. Compute - Fastly products. https://docs.fastly.com/products/compute, 2024.

[19] Floodlight Contributors. Floodlight openflow controller (OSS). https://github.com/floodlight/floodlight, 2024.

[20] Philipp Gackstatter, Pantelis A Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *IEEE International Symposium on Cluster, Cloud and Internet Computing*, 2022.

[21] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with OpenLambda. In *USENIX HotCloud Workshop*, 2016.

[22] Erik Hjelmvik. Hands-on network forensics. https://www.first.org/conference/2015/program#phands-on-network-forensics, 2015.

[23] ICS Lab. The security lab. https://cs3sthlm.se/ics-lab/, 2020.

[24] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[25] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly as an enabler for next generation serverless computing. *Journal of Grid Computing*, 21(3):34, 2023.

[26] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[27] Shuwen Liu. A serverless sdn controller. https://github.com/shuwenliu96/serverless_sdn_controller/, 2025.

[28] Shuwen Liu, Joseph P Petitti, Yunsen Lei, Yu Liu, and Craig A Shue. By your command: Extracting the user actions that create network flows in Android. In *IEEE International Conference on Network of the Future (NoF)*, 2023.

[29] James McCauley. Pox. https://github.com/noxrepo/pox, 2024.

[30] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *IEEE ICDCS Workshops*, 2017.

[31] Mininet. Cbench. https://github.com/mininet/oflops/tree/master/cbench, 2024.

[32] Bong-Hwan Oh, Serdar Vural, Ning Wang, and Rahim Tafazolli. Priority-based flow control for dynamic and reliable flow management in SDN. *IEEE Transactions on Network and Service Management*, 15(4):1720–1732, 2018.

[33] Open Networking Foundation. Open network operating system (ONOS) SDN controller. https://opennetworking.org/onos/, 2024.

[34] perf developers. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2024.

[35] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. In *ACM European Conference on Computer Systems (EuroSyS)*, 2020.

[36] Miroslav Stampar. Daily feed of bad IPs. https://github.com/stamparm/ipsum, 2024.

[37] Curtis R Taylor, Tian Guo, Craig A Shue, and Mohamed E Najd. On the feasibility of cloud-based SDN controllers for residential networks. In *IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2017.

[38] Ivan Vaccari, Giovanni Chiola, Maurizio Aiello, Maurizio Mongelli, and Enrico Cambiaso. MQTTset, a new dataset for machine learning techniques on MQTT. *Sensors*, 20(22):6578, 2020.

[39] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, 11:233–247, 2017.

[40] WasmEdge Contributors. Wasmedge. https://wasmedge.org/, 2024.

[41] Webassembly Community Group. Webassembly. https://webassembly.org/, 2024.

[42] Patrick Wendell and Michael J Freedman. Going viral: flash crowds in an open CDN. In *ACM SIGCOMM Internet Measurement Conference*, pages 549–558, 2011.