QFaaS: Accelerating and Securing Serverless Cloud Networks with QUIC

Kaiyu Hou
Northwestern University, USA
kyhou@u.northwestern.edu

Sen Lin
Northwestern University, USA
sen.lin@u.northwestern.edu

Yan Chen
Northwestern University, USA
ychen@northwestern.edu

Vinod Yegneswaran
SRI International, USA
vinod@csl.sri.com

ABSTRACT

Serverless computing has greatly simplified cloud programming. It liberates cloud tenants from various system administration and resource management tasks, such as configuration and provisioning. Under this new cloud computing paradigm, a single monolithic application is divided into separate stateless functions, *i.e.*, *function-as-a-service* (FaaS), which are then orchestrated together to support complex business logic. But there is a fundamental cost associated with this enhanced flexibility. Internal network connections between functions are now initiated frequently, to support serverless features such as agile autoscaling and function chains, raising communication latency. To alleviate this cost, current serverless providers sacrifice security for performance, keeping internal function communications unencrypted.

We believe that the emerging QUIC protocol, which has secured and accelerated HTTP communications in the wide area, could proffer a solution to this challenge. We design a QUIC-based FaaS framework, called QFaaS, and implement it on the OpenFaaS platform. Our design explicitly ensures that existing serverless applications can directly benefit from QFaaS without any application code modification. Experiments on synthetic functions and real-world applications demonstrate that QFaaS can reduce communication latency for single functions and function chains by 28% and 40%, respectively, and save up to 50 ms in end-user response time.

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563458

CCS CONCEPTS

• **Networks** → **Cloud computing**; *Transport protocols*; *Network performance analysis*; Security protocols.

KEYWORDS

Serverless computing, Cloud networking, Serverless network performance and security, QUIC protocol

ACM Reference Format:
Kaiyu Hou, Sen Lin, Yan Chen, and Vinod Yegneswaran. 2022. QFaaS: Accelerating and Securing Serverless Cloud Networks with QUIC. In *Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3542929.3563458

1 INTRODUCTION

The rapid evolution of lightweight virtualization technology has spawned the rise of the serverless cloud computing paradigm [11, 17, 34]. In serverless computing platforms, cloud providers assume responsibility for all server-related management tasks, including both hardware resource allocation and software runtime preparation. Cloud software developers are thus free to simply focus on designing small discrete stateless functions and orchestrating them together for their high-level business logic.

Among the major allures of serverless computing is *agile autoscaling*. It allows service providers to quickly launch new function instances in response to end-user requests, while saving operational costs. Since auto-scaled instances can be quickly destroyed by cloud providers, tenants only pay for the actual function execution time and do not need to reserve resources for burst requests. Because of both efficiency and economic advantages, serverless computing garners extensive attention from industry and is expected to become the dominant cloud computing paradigm [34]. Its market share is projected to surpass $21 Billion by 2025 [47].

Serverless computing is fundamentally a *network-based* cloud computing paradigm. Thus, optimizing the performance and security of serverless networking is arguably as crucial as existing research efforts on other aspects such

as performance optimization of serverless platforms [1, 15, 33, 56, 66, 69, 78, 80] and security management of cloud functions [3, 23, 64]. Furthermore, the *zero trust* security model has gained considerable momentum among cloud security communities [48, 63]. Under this principle, any entities, even within the same internal network, should not be trusted by default. Therefore, fully encrypting all internal connections is now the best practice for major cloud providers [8, 29, 51]. Although initiating reliable transportation and encryption introduces extra delays, it is not the dominant performance bottleneck in traditional cloud computing: (*i*) transmission delay within the data center is negligible compared to execution times; (*ii*) connection setup latency of TCP and TLS can be simply mitigated by using persistent connections.

However, many leading commercial serverless providers and open-source serverless frameworks still use bare (unencrypted) TCP connections between functions, leaving a potential attack surface [1, 10, 30, 52]. This is due to new challenges that arise specifically in serverless networks. First, with serverless computing, a function instance can be initialized in milliseconds (less than 125 ms for cold-start on AWS [1]) and only processes a small sliver of the computational task (849 ms median execution times on Azure [65]). The latency introduced by TCP and TLS handshakes, even at the sub-millisecond-scale, should no longer be ignored [33]. Second, with the *scale-zero-to-infinity* feature [17], function instances are quickly scaled up and down by cloud providers. It is thus tough to maintain persistent connections between ephemeral functions. Third, as serverless functions are commonly chained together to form task-specific workflows [64], cumulative handshakes exacerbate the end-to-end latency.

In this paper, we raise the following question: *Can we seamlessly enable secure and accelerated network communications for serverless cloud applications?* To address this question, we design and implement a novel solution based on the emerging QUIC protocol, called QFaaS, which can simultaneously improve performance and provide security to existing serverless platforms without the requirements of any tenant code modification.

QUIC [60] is a new transport protocol that has steadily gained popularity on the wide-area Internet [77], particularly for web and video streaming [39]. QUIC combines the advantages of both TLS 1.3 and UDP to provide a secure and reliable transport layer with 0-RTT (round-trip time) connection setup cost, *i.e.*, data packets may be sent without an explicit handshake. QUIC has also been successfully extended to some other scenarios, such as IoT meshes [26, 38], satellite communications [75], and Tor transports [12, 13]. Due to the inherent advantages of reduced handshake costs while providing a secure network, it is appealing to adapt this new protocol to securely address communication performance bottlenecks in emerging serverless networks.

Contributions. Our paper proceeds by first providing a network-centric view of serverless applications, filling in missing details about actual network flow in the widely used logic view. This inspired the design of our QFaaS system, where the QUIC protocol can be seamlessly integrated into serverless platforms, to mitigate connection setup overheads and provide secure communications. Our design explicitly ensures that existing serverless applications can be migrated to QFaaS without any code modification. In addition, serverless applications can be further accelerated by using our function chain library and always-on 0-RTT design. (§3)

We implement the QFaaS prototype into OpenFaaS, the most popular open-source serverless platform, and the system is designed to be easily extensible to contemporary commercial and open-source serverless platforms. The entire system code is made publicly available. (§4)

Our experimental highlights include: *(i)* QFaaS can reduce the single function and function chain response latency by 28% and 40% respectively compared with the state-of-the-art serverless platforms. *(ii)* Upon deploying real-world serverless applications to QFaaS, the end-user response time reduction is up to 50 ms. *(iii)* In certain scenarios, QFaaS was even faster than other platforms using only insecure TCP connections. (§5)

2 BACKGROUND AND MOTIVATION

We provide the background of serverless computing in §2.1. We then discuss the necessity of internal connection encryption and the status quo of serverless computing in §2.2. §2.3 demonstrates that the connection setup latency is a new challenge for serverless networks, which motivates our research.

2.1 Serverless Computing

In serverless computing, traditional applications are decomposed into small code slices, called *functions*. These *stateless* functions then can be orchestrated by tenants to perform their high-level business logic, which is also known as the *function-as-a-service* (FaaS) model. In comparison with the *infrastructure-as-a-service* (IaaS) model, in FaaS, it is no longer incumbent upon the tenants to manage the life cycle of virtual machines (VMs) or the deployment of software stacks. Cloud platform providers undertake all server-related tasks, such as launching VMs, provisioning container clusters, and preparing programming runtimes. From the tenants' perspective, the cloud development and deployment tasks are not server-centric, and thus called "serverless".

The rapid evolution of virtualization technology, especially container technology, is the basis for the emergence of serverless computing. The FaaS model introduces a novel capability to cloud computing: agile auto-scaling without explicit tenant provisioning. Specifically, stateless functions are

usually deployed in lightweight virtualized environments, such as containers. They can be initialized within just a few milliseconds. Therefore, serverless providers can quickly scale the number of running function instances in response to changes in incoming request patterns, in a manner that is automatic, continuous, and completely transparent to tenants. This feature also provides certain inherent economic advantages. Since the allocated resources can be released soon by the cloud providers when they are not in use, tenants will not be charged for idle time and only pay for actual function execution time, *i.e.*, *billing-based-on-usage* model. In effect, the price of handling one thousand burst requests in parallel is roughly the same as the price of handling one thousand requests at low density.

Meanwhile, *backend-as-a-service* (BaaS) is another important component of serverless computing [34]. Cloud providers can provide *stateful* services, such as data storage, event logging, and identity management, to cater to the rapid development of serverless applications.

2.2 Connection Encryption for Zero Trust

Cyber threats emanate not only from tenacious attackers outside the data center but also other insidious entities sharing infrastructure within the same data center. Due to the lack of encryption on internal communication, early-stage data centers exposed extra attacking surfaces to adversaries, leading to several publicized security disasters [36, 62]. Consequently, the zero trust security model came into being. In the zero trust model, no entities should trust each other by default; hence authentication and encryption are always desired. This model has gained popularity in most cloud computing paradigms, such as IaaS and newly-emerging microservices, which dedicates TLS encryption to all connections to be the best practice [8, 29, 51].

Nevertheless, even in those leading commercial serverless platforms, due to the network performance restriction (detailed in §2.3) and the early stage of development, the lack of traffic encryption between internal serverless function communications is still the status quo.

For instance, dedicated traffic encryption is provided by default to most services within AWS data centers [7, 8]. However, for the Lambda serverless computing service [5], AWS only provides traffic encryption in connections between end-users and the Lambda function invoker. The connections between the function invoker and function workers remain unencrypted [1, 10]. Other popular commercial providers, such as Google Cloud and Azure, also do not encrypt the function invoker to function worker connections. In addition, Azure does not require encryption in the gateway to function invoker connections [52] and Google Cloud even allows end-users to trigger serverless functions through the gateway via insecure HTTP requests [30].

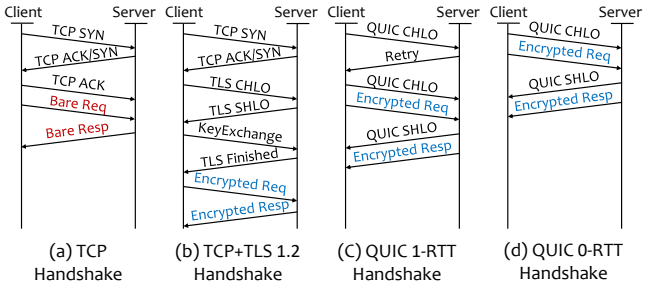
Traffic encryption is also not prevalent in open-source serverless platforms. For example, OpenFaaS [42], the most popular open-source serverless platform, disables all traffic encryption by default. Users must manually enable TLS 1.2 encryption in OpenFaaS, which will, however, significantly impact its network performance, as we show in the evaluation (§5).

Virtual Private Cloud (VPC) is the prevalent solution for IaaS network security. It provides a virtual isolated networking environment in public clouds. However, VPC has several drawbacks when applied to serverless platforms. First, the initialization performance of VPC cannot meet the rapid scaling requirements of serverless computing environments. For instance, the initialization of AWS VPC interfaces takes 15 to 90 seconds, and this cost has to incur per cold-start serverless function call. With this realization, AWS disables tenant VPC for Lambda by default, and *hyperplane VPC interface sharing* was recently announced; however, that still incurs a one-second overhead [6]. We make the case that such delays are prohibitive for serverless functions that initialize and execute at millisecond scales. Additionally, the economic advantages of serverless computing come from efficient hardware multiplexing among tenants. However, exploiting this architectural flexibility limits opportunities for pre-binding tenant VPCs to hardware resources. Finally, VPC is arguably targeted more toward traffic *isolation* than *encryption*. Specifically, cloud providers typically enforce traffic encryption between VMs (*e.g.*, AWS EC2) in the same VPC but not other services due to the hardware and design restrictions [8, 29]. Therefore, VPC is not the off-the-shelf solution for serverless encryption.

2.3 Connection Setup Latency: New Challenge

Internal communication delay was not a significant problem in traditional monolithic applications. Threads within the same process shared the same view of virtual address spaces and inter-process communication (IPC) provided convenient mechanisms (*e.g.*, signals, pipes, and shared memory) for different processes to exchange data efficiently. Cloud computing paradigms increased flexibility in application design and deployment, by breaking up monolithic application stacks into independent services. Nevertheless, there was often some additional cost associated with such flexibility.

In practice, disparate services are commonly deployed in isolated VMs for ease of management. Internal messages between them now must go through a complete network stack, raising communication latency. Such distributed service orchestration also introduces connection setup latency.



Scheme	TCP	TCP + TLS 1.2	QUIC 1-RTT	QUIC 0-RTT
New Session	1	3	1	-
Recover Session	1	2	1	0

Figure 1: Round-trips in different transport protocols: (a) insecure TCP incurs 1 extra RTT; **(b)** in TCP+TLS 1.2 scheme, the encrypted request is sent after 3 RTTs; **(c)** in QUIC 1-RTT mode (new session establishment), the encrypted request is sent after 1 RTT; **(d)** in QUIC 0-RTT (session resumption), the encrypted request is sent immediately.

Specifically, TCP and TLS are used to provide reliable and secure connections between VMs. Both of them require extra round-trips when initiating new connections, as shown in Figure 1 (a) and (b). But such initialization delays were not a severe drawback in cloud computing until the advent of serverless computing. First, maintaining persistent connections among services can partially mitigate the connection setup latency. Though this solution cannot eliminate the delay after launching a new VM, the connection setup latency is still negligible when compared with the VM initiation time. Second, TCP, the dominant protocol for reliable network communication, has been ossified into the OS kernel and is not easily replaceable. Thus, other problems in cloud computing, like scheduling, elastic scaling, and storage, were prioritized over optimizing connection setup latency.

In contrast, the *scale-zero-to-infinity* feature of serverless exponentially magnifies the disadvantages of connection setup latency. To be specific, in serverless computing, *(i)* since the initiation and execution times for function instances are minuscule, the connection setup time is no longer negligible. *(ii)* In addition, the number of running function instances rapidly scales up and down in response to the request changes. It is now not possible to maintain persistent connections between these stateless function instances. *(iii)* Finally, as functions are usually chained together to form task-specific workflows, connection setup costs are incurred at each hop of the function chain, significantly compounding the non-negligible delay. To address these challenges, always keeping at least one instance of each function alive could be a compromise solution (which became an option on AWS Lambda recently). However, such a solution largely increases the tenants' cost and violates the serverless philosophy to some extent. Meanwhile, it still suffers from burst requests. We believe that there is now a greater urgency to prioritize the optimization of connection setup latency in cloud network communications.

3 QFaaS: SYSTEM DESIGN

QFaaS leverages the emerging QUIC protocol to accelerate and secure serverless computing. §3.1 introduces QUIC and emphasizes its benefits for serverless. We then provide a clear network abstraction for serverless applications to identify potential network bottlenecks in §3.2. We describe the system architecture of QFaaS in §3.3, which requires no code modification for existing serverless applications. Designs in §3.4 and §3.5 can further accelerate serverless networking.

3.1 QUIC Protocol for Serverless Networks

QUIC has been quickly and widely adopted on the wide-area Internet after demonstrating the ability to mitigate several drawbacks of TCP (*e.g.*, performance, evolvability). After 2017, more than 7% of Internet traffic (a major part of Google's egress traffic) is under QUIC [39]; in 2021, 5.1% of all websites over the world are using QUIC [77]. QUIC has been standardized by IETF (Internet Engineering Task Force) in RFC 9000 in May 2021 [60]. Furthermore, IETF has just standardized HTTP/3, "HTTP over QUIC", as the next generation HTTP protocol in June 2022 [14]. With rising demand for low latency applications, QUIC gains growing popularity and is likely to outpace TCP on the Internet in the near future.

We think that QUIC can likewise evolve communications in serverless computing as it provides a robust pathway to improve security and performance. On the one hand, cloud users seamlessly benefit from the security of QUIC as it is coupled with the latest TLS 1.3 protocol to provide always-on encryption by design. On the other hand, QUIC can achieve 0-RTT shaving both transport and cryptography handshakes, meaning the first encrypted data packet could be sent before any handshake happens. First, QUIC mitigates the handshake overhead in the TCP protocol, as it provides a reliable multiplexing transport on top of UDP instead of TCP. Second, QUIC further leverages the 0-RTT resumption feature in TLS 1.3. Consequently, QUIC only requires 1 extra round-trip (1-RTT mode) to set up the connection when the client never connected to the server before (Figure 1 (c)). The first encrypted data packet can be sent immediately (0-RTT mode) if the client cached the server information in previous connections (Figure 1 (d)). Thus, QUIC has the potential to greatly reduce the connection setup latency in serverless computing, especially when new function instances are instantly scaled up and chained together to support burst requests.

We notice that QUIC is even better suited for serverless computing environments than the wide-area Internet in some respects. First, enabling QUIC requires modifications on both client- and server-side software to install relevant libraries with compatible versions, which is challenging when the two sides are controlled by different entities. In serverless computing, as cloud providers prepare all the software stacks, including networking-related stacks, software compatibility is no longer an issue. We can further leverage this capability to ensure the always-on 0-RTT (§3.5). Second, the QUIC performance is suffering from the UDP traffic throttling by public Internet service providers. Nevertheless, this is less of a concern within data centers. Third, the 0-RTT replay attack is another concern when using QUIC on the Internet [16]. If it is possible to monitor connections and sniff packets in the middle, the adversary can potentially resend the first client packet to trigger the related request twice on the server-side. Though, performing this attack requires strict conditions, which are harder to achieve inside a data center, we further implement a more secure QFaaS prototype by sending all non-idempotent requests through 1-RTT to mitigate the threat of the 0-RTT replay attack (§4.2).

In addition to security and low latency, using QUIC in serverless networks can also provide many of the same benefits as using it on the Internet. For example, QUIC supports stream multiplexing in one connection. It avoids head-of-line blocking delays due to the TCP's sequential delivery. Besides, as QUIC runs in the user space instead of the kernel, the transport layer is now more malleable to meet evolving application demands with frequent updates, such as a notable recent breakthrough: *Pluginized QUIC* [24].

TCP Fast Open (TFO) [79] is a potential competitor to QUIC [18]. It allows the application data attached to the client SYN packet to avoid the TCP handshake latency if the SYN packet contains an identifier (TFO cookie) from the last connection. Though TLS 1.3 (0-RTT) over TFO theoretically provides the same round-trip performance as QUIC, because of several deep-rooted privacy and performance issues, TLS over TFO has been disabled on all modern browsers (*e.g.*, Chrome, Firefox, and Edge) and is not yet actively used by most popular operating systems after 10 years [70]. First, TFO relies on a unique unencrypted cookie in the TCP header, which leads to severe tracking concerns on the public Internet. In addition, enabling TFO requires updating all middleboxes in data centers, such as firewalls, proxies, and security devices, to support a non-originally designed TCP option. Nevertheless, these network core devices are ossified in the network and rarely updated. Consequently, failed TFO requires an ordinary TCP SYN retry, leading to TFO actually increasing round-trips. In contrast, QUIC runs over UDP and only requests updates on end devices. Finally, only messages which are smaller than the MTU (*i.e.*, those that may be embedded in the TCP SYN payload) will benefit from the TFO, while the entire client-initialized message (spanning multiple packets) can benefit from QUIC's 0-RTT feature. Therefore, we advocate for QUIC over TFO in the QFaaS design.

3.2 Modeling Serverless Networks

To use QUIC in serverless computing, the first challenge is to identify network connections in its architecture. Though the logic abstract model of serverless platforms is commonly used, important details were missed with respect to the network modeling. We address the limitations by providing a new abstraction of the serverless architecture through the network-centric view. This model will guide our QFaaS system design.

Logic Model. General discussion about serverless architectures is commonly framed in the context of the logic view, shown in Figure 2 (a). Under this abstraction, a unified API Gateway continuously listens for end-user requests. Upon receiving a function invocation, Gateway first executes permission authentication and scales corresponding function instances. Gateway then forwards the user request to a function instance behind it. Functions chained together with backend services compose an integrated serverless application and perform cloud tenants' business logic.

While the logic model largely simplifies the connection details, such that one can quickly understand the essential concepts of serverless computing, it does not reflect actual network flows. There are two important details missed: *(i)* after the end-user sends a function request to Gateway, the response of this function (F_A or F_C) is returned through Gateway instead of directly by the function; *(ii)* in function chains, when a function (*e.g.*, F_A) sends a request to another function (F_B), this request must also go through Gateway. Because only Gateway can launch new instances of functions, and knows the destination address of their running instances.

Network-centric Model. To address the aforementioned limitations, we provide a new abstraction of the serverless architecture through the network-centric view (Figure 2 (b)). In this model, the serverless architecture is divided into two parts: the gateway subsystem and the workers subsystem.

- Components in the gateway subsystem expose static function interfaces to end-users, manage running workers, and dispatch requests to corresponding functions. These services are all *stateful* and run on *permanent* machines. In existing serverless platforms, corresponding modules may have variant names. For example, in AWS, they are called frontend and worker manager; in OpenFaaS, they are called `api-gateway` and `faas-netes` controller. Regardless of the names, they provide the same functionality.

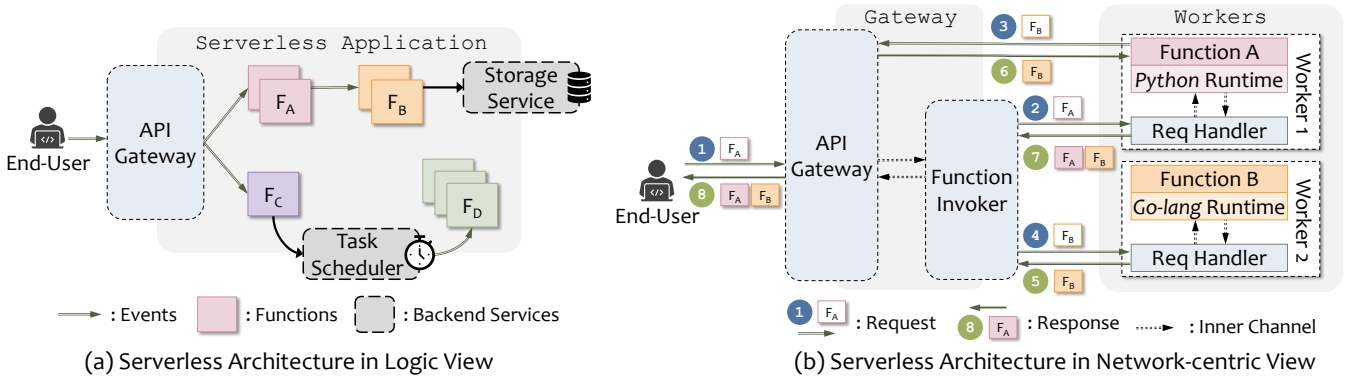


Figure 2: Serverless Architecture. (a) Logic Model. Gateway forwards end-user requests to corresponding functions. Functions chained together with backend services compose a serverless application. However, network details are missing: *(i)* function results are returned through Gateway to end-users, *(ii)* functions are chained through Gateway, *i.e.*, no direct connections between end-users and ephemeral function instances, and between two instances. **(b) Network-centric Model.** Gateway components run permanently, expose function interfaces, manage running workers, and dispatch requests. Workers run on ephemeral containers, host request handlers and different language runtime for functions. The request handlers can only be connected by Gateway.

- Workers are *ephemeral* containers that comprise the request handler, the function runtime, and tenant function code. The request handler provides the internal communication ability for workers. It receives trigger requests from Gateway and sends function results back to Gateway. The function runtime provides isolated software stacks and programming language libraries to execute tenant function code. Therefore, tenant functions are decoupled from the management of ingress network connections. Hence, the request handler can be designed independently by serverless providers.

Figure 2 (b) shows an example where an end-user requests the service of function F_A, while F_A chained together with F_B provides the service to the end-user. In this example,

- (❶|❽) the end-user sends F_A a request (❶) and receives responses of $F_A|F_B$ (❽) from the direct connection with API Gateway. In the process, Gateway acts as a transport layer *server*, listening and responding to user requests. Network details behind it are transparent to the end-user.
- (❷|❼) API Gateway forwards the F_A trigger event to Function Invoker. After the F_A worker is initialized, Function Invoker sets a connection to the request handler in F_A worker, sends request data (❷), and receives responses (❼). In this process, Function Invoker plays the role of the transport layer *client* to initiate this connection. The request handler plays the role of the transport layer *server*.
- (❸|❻) F_A needs the response of F_B. Nevertheless, instead of sending a request directly to a worker of F_B, F_A will send the F_B request (❸) and receive responses of F_B (❻) from Gateway. F_A does not need to care about any scheduling details of F_B. In this process, Gateway acts as a transport layer *server* again, even though the connection is internal.
- (❹|❺) Function Invoker initializes a worker for F_B, sets up a connection to its request handler, sends request data (❹), and receives responses (❺) from this connection.

In current serverless platforms, HTTP (including REST-API and gRPC) is commonly used application layer protocols for connections to API Gateway (❶|❽, ❸|❻) and request handlers (❷|❼, ❹|❺). These application layer protocols rely on TCP and TLS protocols underneath to provide reliable and secure transport communications. For security concerns, connections involved API Gateway (❶|❽, ❸|❻), which exposes interfaces to outside, are mandatorily encrypted by most providers (§2.2). For other connections in Figure 2 (b), the data exchange between request handlers and functions in the same worker is through IPC with negligible overhead. API Gateway and Function Invoker are usually deployed in different machines. But they can maintain a persistent connection to mitigate the connection setup overhead.

3.3 QFaaS System Architecture

We first identify connections that affect the serverless network performance and can be seamlessly optimized without tenants' code modification. In our network-centric view, API Gateway to Function Invoker connections and request handler to language runtime connections could be persistent or through IPCs. The connection between end-users and

Gateway (❶|❽) is initialized by end-users and could also be persistent. The connection from the function to Gateway (❸|❻) is initialized by functions. In contrast, the connections from Gateway to workers (❷|❼, ❹|❺), which are fully controlled by providers, expose opportunities to optimize serverless networks.

First, function workers are instantaneously launched and torn down in response to requests. We cannot simply use persistent connections to mitigate the connection setup latency for function workers. Second, this overhead will be multiplied when functions are chained together or the number of running instances is quickly scaled up. Thus, these serverless-introduced bottlenecks drive major cloud providers to sacrifice security for performance, keeping ❷|❼, ❹|❺ unencrypted (§2.2).

To accelerate serverless networking while maintaining security, we introduce the design of QFaaS, as shown in Figure 3. In this design, we integrate the QUIC client into Function Invoker and also integrate the QUIC servers into the worker request handlers (to replace the TCP and TLS client and servers, respectively). All function requests that go through Gateway to workers would now benefit from the efficiency and security of the QUIC protocol.

On the one hand, QUIC embraces full encryption by default and employs the latest TLS 1.3 protocol. As a result, connections from Gateway to workers would benefit from security improvements provided by TLS 1.3. On the other hand, this QFaaS design can ensure the activation of the QUIC 0-RTT feature. To enable 0-RTT connection resumption, QUIC leverages *QUIC connection tokens* and *TLS session caches* stored on the client-side. In serverless architecture, Gateway runs on stateful machines and plays the role of the QUIC client in QFaaS. We integrate the *QFaaS 0-RTT Store* into the Gateway to maintain and manage connection-specific information. Therefore, serverless applications under this design can further benefit from the 0-RTT feature.

Moreover, QFaaS design does not request any changes to tenants' function code. In serverless computing, all running containers, as well as the code of Gateway and request handlers, are provided and controlled by cloud providers. Modifications are transparent to cloud tenants and end-users.

3.4 Function Chain Library

An inquisitive reader might wonder why we did not further replace the connection initiated from the function to Gateway (❸|❻) with QUIC? This is because such a connection is function code related and programming-language specific. Specifically, Gateway usually exposes URLs or REST APIs for functions. End-users can invoke a function by sending an HTTP request to the corresponding URL. Similarly, when a tenant wants to invoke a function (F_B) by another function

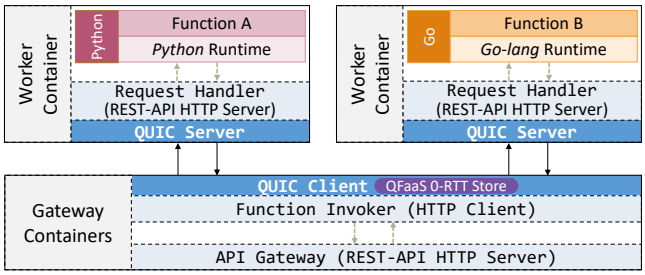


Figure 3: System Design of QFaaS. QUIC client and QUIC servers are integrated into Function Invoker and worker request handlers to replace the TCP/TLS client and servers. This modification is transparent to cloud tenants and ensures the activation of the QUIC 0-RTT feature.

(F_A) to form a function chain, the tenant also must initiate an HTTP request by the function code in F_A and follows the programming language practices. For example, AWS, Azure, Google Cloud, and OpenFaaS all suggest Python users form function chains by leveraging the Python `Requests` library [35]. Therefore, such connections are not fully controlled by the cloud providers and cannot be optimized as described in §3.3.

One alternative way to enable QUIC at ❸|❻ is to provide a QUIC server at Gateway and ask developers to integrate a QUIC client in their function code. Nevertheless, this design requires significant code modification and also requires developers to be familiar with QUIC client configurations.

Recent advancements in platform specific libraries allow for an improved design. For fine-grained access control and ease of use, some serverless platforms, such as AWS Lambda, now provide libraries under different languages for tenants to form function chains [9]. With such libraries, developers can directly call platform APIs to invoke another function instead of explicitly sending an HTTP request by the code.

Leveraging this idea, we provide a QFaaS function chain library to enable QUIC at ❸|❻, requiring slight tenant code modification. This chain library has QUIC as its underline transport layer protocol. We integrate the QUIC server into Gateway to accept function requests through QUIC. Thus, all function chain traffic invoked by the library will benefit from the short latency and security of QUIC. Currently, this library supports Python3 and Go-lang, which are two popular programming languages used in all major serverless platforms. The code modification to adapt this library is minimal. For instance, the Python developers only need to import the library and switch their `Requests` call to the QFaaS chain library call, which are only 2 lines of code modification.

This design also has three side benefits. First, Gateway now has the ability to accept QUIC requests. It is now possible for end-users to initiate a request by QUIC and further

accelerate the ❶|❽ connection. Second, this new ability will not interfere with existing TLS Gateway functionalities, as QUIC listens on the UDP port while TLS listens on the TCP port. Third, the system can now be more easily integrated with current serverless security and access control mechanisms [3, 23, 64].

3.5 Always-on 0-RTT QUIC

QUIC is initially designed for the wide-area Internet, where connection peers are controlled by different entities. On the contrary, in serverless networks, providers can fully administrate the platform. Leveraging this ability, we propose the *Always-on 0-RTT QUIC* design, which ensures the activation of 0-RTT even for completely cold-start functions.

In the QUIC protocol, if the client has never connected to the server, the first request will use 1-RTT mode, due to the lack of pre-knowledge with the server. QUIC clients rely on the QUIC connection token and TLS session cache from previous handshakes to enable 0-RTT. The QUIC connection token is used for servers to identify and verify the 0-RTT connection from clients. The TLS session cache is indeed the TLS pre-shared key (PSK) [25, 44], which is the basis for 0-RTT encryption.

We introduce a *QFaaS 0-RTT Generator* component in the QFaaS design. When launching a cold-start worker (QUIC server), the *Generator* will put a valid Function Invoker (QUIC client) token into the worker, so the 0-RTT connection from Function Invoker can be accepted. In addition, the *Generator* will also produce a unique PSK and insert it into both sides, which will then be used for 0-RTT encryption. As this process is a part of workers' environment setup, it will not introduce extra delay. After the handshake process is complete, the server will provide a new token and session to the client for the next 0-RTT connection using the QUIC protocol. These will be stored in the *QFaaS 0-RTT Store* (§3.3). This design utilizes the advantages of serverless computing and is fully compatible with the QUIC protocol.

4 QFaaS: SYSTEM IMPLEMENTATION

We implemented the QFaaS prototype on OpenFaaS (§4.1) and enabled the QUIC 0-RTT feature on it (§4.2). The QFaaS design is easy to be extended to other platforms (§4.3).

4.1 QFaaS Prototype on OpenFaaS

We implemented our QFaaS into the popular OpenFaaS [42]. OpenFaaS is currently the leading open-source serverless platform (sorted by GitHub stars [57]). It uses Docker containers to host all components and Kubernetes (K8s) to simplify container deployment and management.

We extended `quic-go` [43] for our prototype. `quic-go` supports the recently standardized IETF QUIC [60]. And it is implemented in Go-lang, the same language as OpenFaaS and K8s, which makes them easier to be integrated. `quic-go` also provides an HTTP/3 [14] implementation by assembling QUIC with the Go-lang HTTP package (`net/http`).

We primarily modified two components of OpenFaaS: `faas-netes` and `of-watchdog`.

`faas-netes` is the Function Invoker component in the OpenFaaS platform that resides on Gateway. It controls the life cycle of worker containers by sending commands to the K8s master. It also works as an HTTP client, forwarding function requests to corresponding workers through standard HTTP messages. We modified `faas-netes`, integrating a `quic-go` HTTP/3 client module and coordinating it with the remaining parts. All function requests then are proxied and encrypted by QUIC when they are forwarded to workers. All these modifications only introduce a minimal increase (15 MB) to the size of compiled `faas-netes` container images.

`of-watchdog` is a tiny HTTP server, working as the request handler inside the function worker container. `of-watchdog` uses an internal IP address and is only reachable within the K8s cluster. It receives incoming function requests from `faas-netes` and passes them on to the function. We reformed the HTTP server module in `of-watchdog` to the `quic-go` HTTP/3 server such that it can accept QUIC connections from `faas-netes` and decrypt HTTP/3 messages. After attaching related packages for HTTP/3 and QUIC, the size of `of-watchdog` executable file only increased by 3 MB.

Besides `of-watchdog`, an OpenFaaS worker container also contains a language runtime and the tenant function code. As the runtime is independent to `of-watchdog`, QFaaS inherently support tenant function code in various languages with its one-size-fits-all `of-watchdog` implementation. There is no need to modify a specific language runtime.

To support QFaaS function chain library, we also installed a QUIC server into the OpenFaaS `api-gateway`. It will receive and respond to all function invoke requests from the function chain library and end-users using a QUIC client. This modification does not interfere with exiting Gateway functionalities as it listens on the UDP port.

We have open sourced our prototype implementation.[1] In addition, we are working on integrating QFaaS as a plugin into the OpenFaaS platform and collaborating with a leading cloud provider to enable a proof-of-concept (PoC) deployment of QFaaS in its serverless service.

4.2 QUIC 0-RTT Activation

To further enable the QUIC 0-RTT feature, we implemented the *QFaaS 0-RTT Store* in `faas-netes` to maintain and manage the QUIC connection tokens and TLS session caches for QUIC 0-RTT connections. With this implementation,

[1] https://github.com/qfaas-project

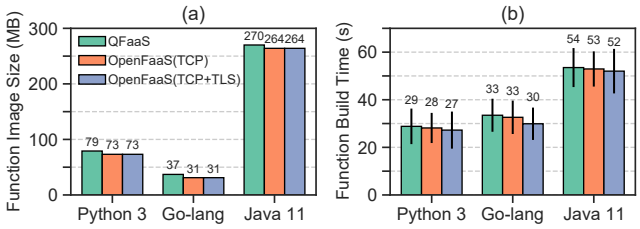


Figure 4: (a) Function image sizes and (b) image build time using different `of-watchdogs` and language runtimes.

scenarios, such as the function warm-start, resuming suspended workers, and processing non-continuous requests, will benefit from the performance of QUIC 0-RTT. We further implemented the *QFaaS 0-RTT Generator* in the K8s control component. It generates and distributes the 0-RTT connection information when a cold-start worker is launching. Thus, the cold-start scenario will be accelerated by 0-RTT.

The replay attack [16] is a major security threat when using QUIC's 0-RTT mode. Under sophisticated settings, an adversary could potentially replay the first 0-RTT message to trigger the corresponding action twice on the server [25]. To be specific, man-in-the-middle (MITM) sniffing and packet replay are two necessary conditions for the QUIC 0-RTT replay attack. As for QFaaS, on the one hand, internal data-center networks have different characteristics than the public wide-area Internet. Both conditions may be harder to achieve inside a data center. On the other hand, to provide a higher security level, following the suggestions of [25, 60], we provided a more secure QFaaS option, which mandatorily sends all non-idempotent [61] requests (*e.g.*, POST) through 1-RTT. This option further mitigates the threat of the 0-RTT replay attack. Users can make choices in QFaaS based on their security needs. Additionally, recent cryptography research [31] also shows that it might be possible to support perfect forward secrecy during the 0-RTT key exchange process. We will show in our evaluation that even when operating in 1-RTT mode, QFaaS is still considerably faster than the OpenFaaS platform because it still requires fewer RTTs than the TCP+TLS scheme.

4.3 Platform Universality

The QFaaS design is not only effective for OpenFaaS but can also be easily extended to other serverless platforms. This is because the network-centric model we provided is universal to prevalent serverless architectures. For instance, network flows in Lambda also follow this model. Specifically, AWS revealed the Lambda architecture in [1]. Unlike Google Cloud or OpenFaaS, AWS Lambda uses microVMs instead of containers for function workers, where each worker also contains a request handler (called λ shim) that listens to HTTP requests from the Lambda Frontend. Thus the QFaaS design can be directly applied in the Lambda architecture, *i.e.*, by integrating the QUIC server and client into the λ shim and Frontend, and maintaining the *QFaaS 0-RTT Store* at Frontend. In addition, Lambda now provides libraries for access control and function chains [9]. QFaaS function chain library can be implemented into it to further accelerate chain communications.

QFaaS can also be easily migrated to other open-source serverless platforms. Taking Apache OpenWhisk [73] as an example, the ❷|❼ and ❹|❺ connections are essentially the connections between OpenWhisk Controller and Code Invoker that we can use QUIC to secure and accelerate. Additionally, we can apply the design of QFaaS function chain library into special *trigger events* supported by OpenWhisk.

5 EVALUATION

We answer the following questions about QFaaS here. What are our testbed and experiment settings (§5.1)? Will QFaaS introduce extra overheads in building function images (§5.2)? How does QFaaS perform on a single function in comparison with TCP and TCP+TLS in different scenarios (§5.3)? How does QFaaS react to variant intra-cloud delays (§5.4)? How does the length of function chains impact QFaaS performance benefits (§5.5)? And how well do the benefits transfer to real-world serverless applications (§5.6)?

5.1 Testbed and Experiment Settings

We evaluate the performance of QFaaS using several synthetic serverless functions and a real-world commercial serverless application *Hello, Retail!* [53, 54]. These synthetic functions are designed to cover different scenarios independently, including simple echo functions, functions with large content data, and function chains with variant lengths [81]. *Hello, Retail!* is a popular open-source serverless application used in many recent serverless studies [3, 23, 55, 64]. We use it to assess the benefits that QFaaS can deliver in the real world.

We compare the performance of QFaaS with respect to OpenFaaS, using TLS 1.2, for inter-component communications. We also measure the performance of OpenFaaS using only bare TCP connections between components as a reference. Because TCP does not encrypt and decrypt packets, it has a much shorter OS processing delay in comparison to secure protocols. However, TCP-only OpenFaaS does not provide any security for the cloud platform. We find that our QFaaS design is even faster than TCP-only OpenFaaS in some scenarios, while providing additional security.

All experiments were performed on our K8s cluster with 1 master node and 3 follower nodes. The K8s system components were deployed in the master node. All OpenFaaS related components were running in the follower nodes and

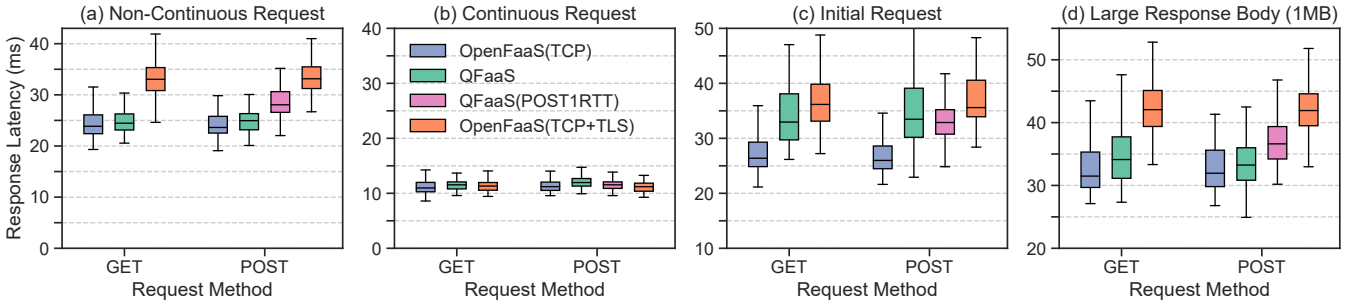


Figure 5: Single function end-user response latency: (a) non-continuous requests require connection resumption. For GET requests, QFaaS shows the same performance as insecure OpenFaaS (TCP) and is 28% better than OpenFaaS (TCP+TLS). For POST requests, QFaaS in its 1-RTT mode is still 14% faster than OpenFaaS (TCP+TLS); **(b) continuous requests** have no connection setup overhead. They all perform identically; when an **(c) initial request** is sent to a newly launched function, if no session caches in Gateway, QFaaS is still 11% faster than OpenFaaS (TCP+TLS); in the **(d) large response** scenario, QFaaS is slightly slower than OpenFaaS (TCP) due to encryption overhead. But it is still 21% faster than OpenFaaS (TCP+TLS).

each node has a 4x2.9 GHz CPU with 8 GB of RAM. All Docker images were pre-pulled to avoid the influence of external network variations. We also enabled the K8s local DNS agent feature to improve cluster DNS performance.

We use the default MTU value of 1500 and keep all TCP, TLS, and QUIC settings to be the default values from standard Go-lang libraries (go1.15). A recent study [71] indicated that QUIC could show better performance by MTU tuning. QFaaS users can also potentially achieve better performance by tuning protocol settings, such as congestion control scheme, generic receive offload (GRO), and generic segmentation offload (GSO) based on their traffic characteristics.

5.2 Function Image Overheads

Using QFaaS does not require any application code modification. To migrate existing OpenFaaS application functions to QFaaS, cloud providers only need to rebuild the function container image on top of the modified `of-watchdog`. This process can be done automatically and is transparent to tenants. The function image size and function image build time overheads are given in Figure 4 (a) and (b), respectively.

Results: Figure 4 (a) indicates that QFaaS only slightly increases the container image size by 3 MB among each different language runtimes. The increased size is due to the additional libraries for QUIC server support. As shown in Figure 4 (b), QFaaS only introduces negligible addition time in building functions comparing with OpenFaaS.

5.3 Single Function Performance

We measure the response latency of several synthetic single functions to show the benefits of QFaaS under different scenarios independently (Figure 5). Response latency represents the time interval between the end-user sending the request and receiving the complete function response. For GET requests, we compare the latency between OpenFaaS (TCP), QFaaS, and OpenFaaS (TCP+TLS). For POST requests, we also measure the latency of QFaaS with mandatory 1-RTT enabled, *i.e.*, QFaaS (POST1RTT) (check §4.2 for more details). In Figure 5 (a), (b), and (c), we all use a simple echo function to avoid the jitter in function execution. The function used in Figure 5 (d) returns a large response body of 1 MB when called. We repeat each experiment 100 times. We use the default intra-cloud delay of 0.5 ms. The QFaaS performance under other delays is detailed in §5.4.

Results: Each box plot in Figure 5 (as well as Figure 10 and Figure 11) depicts the maximum, third-quartile, median, first-quartile, and minimum through dash marks from the top to the bottom.

As shown in Figure 5 (a), we first measure the scenario that end-user requests sending in the no-continuous pattern. In this case, whether TCP, TLS, or QUIC, connection resumption is required. For GET requests, QFaaS performs the same as insecure OpenFaaS (TCP) and is 28% better than OpenFaaS (TCP+TLS). For POST requests, QFaaS working in 1-RTT mode is still 14% faster than OpenFaaS (TCP+TLS) and achieves the same benefits as it performs on GET in 0-RTT mode. The latency difference between QFaaS and OpenFaaS (TCP+TLS) is larger than simply counting extra RTTs because extra handshake packets across protocol stacks at two ends also introduce additional processing delays.

In Figure 5 (b), end-users continuously send requests to avoid any connection resumption. All these implementations perform identically. It indicates that QFaaS does not bring additional overhead for this scenario.

In Figure 5 (c), we measure the first request latency when a function instance is newly launched. In this scenario, we let the QUIC client in `faas-netes` have no server session cache

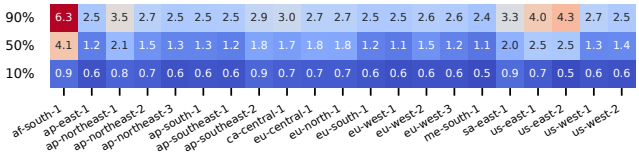


Figure 6: AWS Lambda intra-region latency (ms) in the 10th, 50th, and 90th percentiles in one year. Source data is collected from [49] between June 2021 to June 2022.

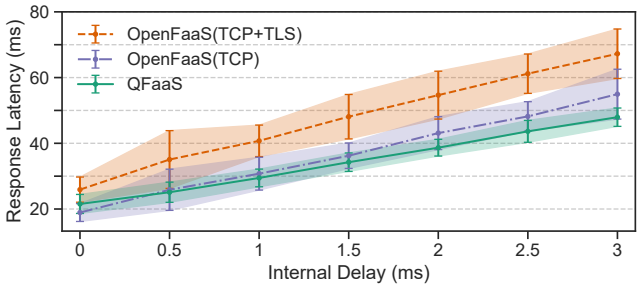


Figure 7: Benefits of QFaaS under variant intra-cloud delays. QFaaS is always faster (17% reduction) than OpenFaaS (TCP+TLS) even when the internal delay is 0. The response latency difference increases as delays increase. QFaaS starts to be faster than OpenFaaS (TCP) when delay>0.5 ms.

corresponding to the new function worker, thus working in 1-RTT mode. In this case, QFaaS is still 11% faster than OpenFaaS (TCP+TLS).

Figure 5 (d) shows the scenario when the function returns a large body. The end-user needs to wait for several packets before getting the complete response. In this scenario, QFaaS is slightly slower than the insecure OpenFaaS (TCP) due to traffic encryption and decryption overhead. But it is still 21% faster than OpenFaaS (TCP+TLS) when the size of the response body is 1 MB.

5.4 Variant Intra-Cloud Delays

The end-user response latency reduction of QFaaS is related to the intra-cloud delays. Because the QFaaS advantage over OpenFaaS is obtained from reducing the number of RTTs in the connection setup. The network delay within a typical data center has been found to be around 0.5 ms in prior studies [32, 41, 59]. In our evaluation, we set the default delay between cluster nodes to 0.5 ms. One may notice that the AWS EC2 ping delay is sometimes less than 0.1 ms. Because AWS tends to deploy EC2 VMs of the same tenants into the same host machine [78]. Nevertheless, this is not the case for serverless computing. As multiple tenants share the same Gateway infrastructure, the delay of ❷|❼ or ❹|❺ connection is closer to the average intra-cloud delay.

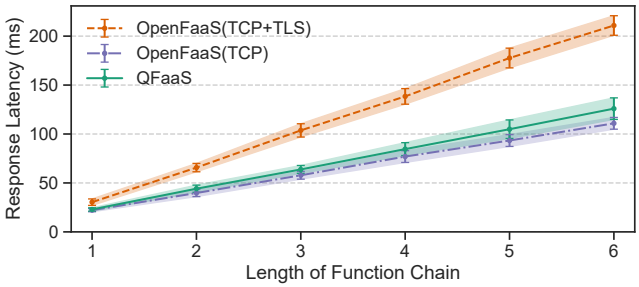


Figure 8: Benefits of QFaaS with the function chain library. The latency difference between QFaaS and OpenFaaS (TCP+TLS) increases as the chain's length increases and reaches 85 ms (40%) when the length is 6.

Figure 6 shows the AWS Lambda intra-cloud latency percentiles in the same region. The majority lies in the range from 0.5 ms to 3 ms. We will show the benefits of QFaaS against these delays. In addition, the internal network delay is also potentially very small. We thus specifically measure the case of zero network delays by deploying all the platform components and function workers in the same host machine.

We show the benefits of QFaaS under different cloud internal delays in Figure 7. The testing scenario is the same as Figure 5 (a). As the internal delay increases, the response latency will also increase. But the latency increase in QFaaS is always smaller than that of its opponents.

Results: The colored regions in Figure 7 (as well as Figure 8) are mean values plus/minus standard deviations. Specifically, when the internal network delay is 0, QFaaS is still 17% faster than OpenFaaS (TCP+TLS). Because QFaaS can still save processing delay costs of RTTs in the protocol stacks compared with OpenFaaS (TCP+TLS). QFaaS maintains its advantages over OpenFaaS (TCP+TLS) as the internal delay increases. Their latency difference reaches 19 ms when the internal delay is 3 ms. QFaaS becomes faster than insecure OpenFaaS (TCP) after the internal delay is greater than 0.5 ms, and is 13% faster than OpenFaaS (TCP) when the internal delay is 3 ms.

5.5 Function Chain Performance

In serverless applications, functions are commonly chained together to perform a complete task flow. We measure the benefits of using QFaaS function chain library in different lengths of function chains in Figure 8. The experiment design follows the nested function chain implementation in ServerlessBench [81]. In a function chain, the end-user invokes the ingress function; one function invokes another function and returns until the invoked function responds. Other settings are the same as in Figure 5 (a).

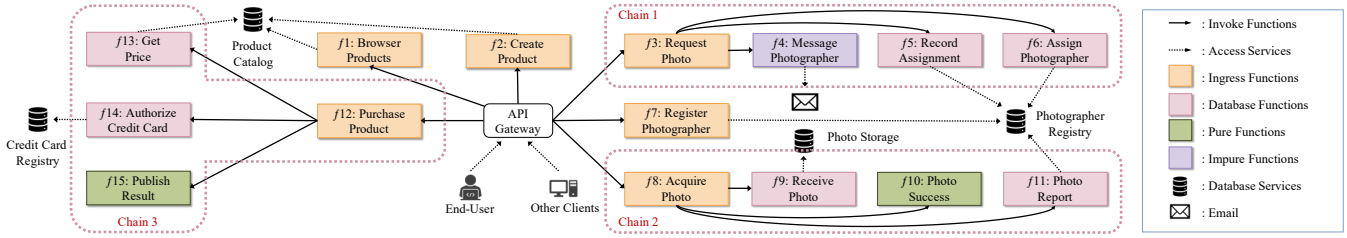


Figure 9: A reference architecture of the *Hello, Retail!* application. Rectangles represent serverless functions, and are categorized into different colors by their attributes. They form 3 major function chains (See Table 1 for details).

Table 1: Evaluated Serverless Function Scenarios in the "*Hello, Retail!*" Application

Scenarios	Function	Function Name	HTTP Method	Function	Function Name	HTTP Method
Pure Functions	$f10$	Photo Success	GET	$f15$	Publish Result	POST
Database Functions	$f1$	Browse Products	GET	$f7$	Register Photographer	POST
Chain Functions	$f3$	Request Photo	POST	*Invoking a function chain:* $f3 \rightarrow f4 \rightarrow f5 \rightarrow f6$.		
	$f8$	Acquire Photo	POST	*Invoking a function chain:* $f8 \rightarrow f9 \rightarrow f10 \rightarrow f11$.		
	$f12$	Purchase Product	POST	*Invoking a function chain:* $f12 \rightarrow f13 \rightarrow f14 \rightarrow f15$.		

Results: As shown in the result, compared with OpenFaaS (TCP), regardless of the length of the function chain, QFaaS always has a similar end-user response latency as the insecure OpenFaaS (TCP). QFaaS always performs better than OpenFaaS (TCP+TLS), and their latency difference increases as the chain's length increases. QFaaS is 85 ms (40%) faster than OpenFaaS (TCP+TLS) when the chain length is 6.

5.6 Real-world Application Performance

Hello, Retail!. To better understand how QFaaS works in production environments, we first conducted experiments on a real-world serverless application *Hello, Retail!*. It implements a functional retail platform constructed by a set of serverless functions and back-end services. Figure 9 shows the reference architecture of *Hello, Retail!*. Please note, this figure uses the serverless logic view instead of the actual network-centric view, to highlight the application's abstract structure. It is a real-world serverless application originally developed by Nordstrom on AWS Lambda. We ported the entire *Hello, Retail!* application to the OpenFaaS platform as described in prior work [23, 64]. It is also deployed into QFaaS without any code modification.

As shown in Figure 9, *Hello, Retail!* consists of 15 functions. These functions form 3 major function chains. Table 1 lists all scenarios we used in our experiments, covering the most representative scenarios in this application. In terms of whether a function accesses backend services and whether it invokes a function chain, we classify the scenarios as:

- **Pure Functions:** The function only communicates with the Gateway.
- **Database Functions:** The function will access back-end services, *e.g.*, database.
- **Chain Functions:** The ingress function that sequences a function chain.
- **Chain Functions with Function Chain Library:** Chain functions adopting the QFaaS function chain library.

Following the preceding experiment setting in Figure 5 (a), we measured the end-user response latency by sending non-continuous function requests. Figure 10 shows the results.

Results: (a) For pure functions, $f10$ is invoked by GET messages, and $f15$ is invoked by POST messages. QFaaS and QFaaS (POST1RTT) can achieve a similar acceleration as they performed in single function evaluations (Figure 5). (b) For database functions, though the performance boosts are diluted by the extra connections with databases or third-party services, QFaaS can still achieve 7%-12% latency reduction against OpenFaaS (TCP+TLS) while keeping comparable performance as insecure OpenFaaS (TCP). (c) For chain functions, QFaaS remains to outperform OpenFaaS (TCP+TLS) by 14%-25% working in 0-RTT mode and 6%-10% working in 1-RTT mode. Additionally, QFaaS bonuses multiply to attain up to 50 ms latency reduction, which would perceptibly improve user experience. (d) To take full advantage of QFaaS, we integrated the function chain library to $f12$. Since the natural language efficiency distinction, where the original *Hello, Retail!* is written in NodeJS and the QFaaS function chain library is implemented in Go-lang, for comparison fairness, we translated origin Chain 3 ($f12$) in Go-lang. It shows a 21% latency reduction. The results on *Hello, Retail!* demonstrate

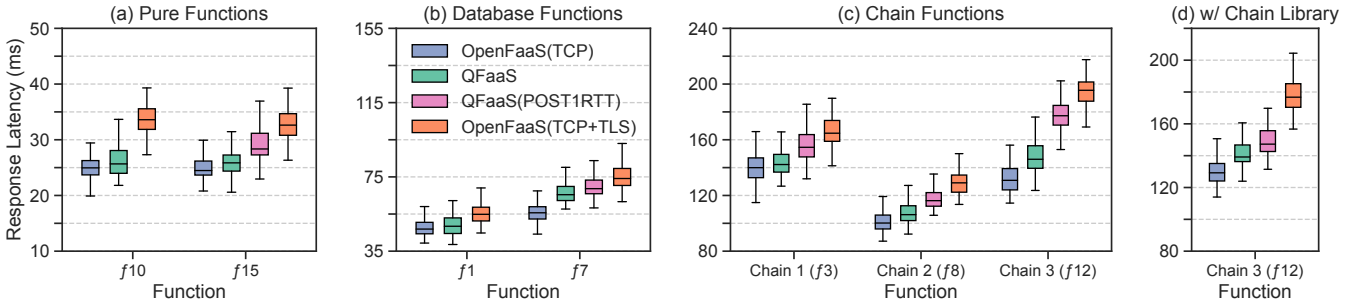


Figure 10: End-user response latency in the *Hello, Retail!* application. (a) Pure Functions: $f10$ accepting GET requests. $f15$ accepting POST requests. QFaaS achieved the same results as in Figure 5. **(b) Database Functions**: $f1$ accepting GET requests. $f7$ accepting POST requests. QFaaS is 12% faster than OpenFaaS (TCP+TLS); QFaaS (POST1RTT) is faster by 7%. **(c) Chain Functions**: 14%-25% latency reduction (up to 50 ms) provided by QFaaS in one request; 6%-10% latency reduction with QFaaS (POST1RTT). **(d) Chain Functions with Function Chain Library**: Chain 3 gaining a 21% performance boost.

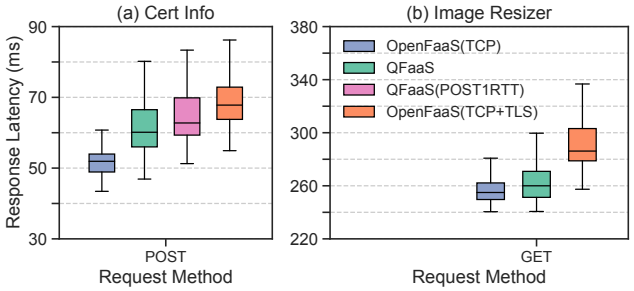


Figure 11: End-user response latency in the *Cert Info* and the *Image Resizer* applications. (a) *Cert Info*: QFaaS demonstrates similar improvements as in Figure 10 (b). **(b) *Image Resizer*:** Compared to OpenFaaS (TCP+TLS), QFaaS saves 26 ms while providing secured communications in image transmission.

that cloud tenants can instantly gain the benefits of QFaaS as synthetic serverless functions (§5.3).

***Cert Info* and *Image Resizer*.** In addition to *Hello, Retail!*, we then evaluated two more open-source serverless applications from *OpenFaaS Function Store* [58]. Both applications are composed of a single serverless function but involve communications to third parties. The *Cert Info* application [68] accepts POST requests and will connect to the wide-area networks. It leverages Go-lang standard libraries to fetch the certificate information associated with the requested domain name from the Internet. The *Image Resizer* application [72] accepts GET requests and demands more computational overhead than previous ones. It first downloads a large image file and resizes it locally. The resized image will be put into the GET response. Evaluation results are shown in Figure 11.

Results: (a) *Cert Info*: Compared to OpenFaaS (TCP+TLS), which needs 16 more ms than OpenFaaS (TCP) to achieve security, QFaaS and QFaaS (POST1RTT) only require half the extra time, *i.e.*, 8 ms and 10 ms, respectively. (b) *Image Resizer*: QFaaS provides comparable end-user latency to OpenFaaS (TCP) and is 26 ms faster than OpenFaaS (TCP+TLS).

Through our evaluations on real-world serverless applications, we believe serverless computing platforms will be more attractive in miscellaneous workloads for existing cloud applications when the low-latency requirement can be met.

6 RELATED WORK

We present related research efforts of serverless computing in §6.1 and the evolution and extensions of QUIC in §6.2.

6.1 Serverless Computing Research

The rising prominence of serverless computing has attracted recent research interests in wide-ranging topics. We summarize related work in the following closely-related categories: security and access control, virtualization optimization, scaling and scheduling, and performance benchmarking.

Security and Access Control. Trapeze [3] uses a language-based dynamic information flow control (IFC) to secure serverless functions. Each serverless function in Trapeze is wrapped by a security IFC shim to share data stores and exchange messages. Valve [23] employs function level flow control to restrict unexpected function behaviors through the network. WILL.IAM [64] encodes absolute and conditional information flows into a graph to disallow access policy violations at the ingress. Nevertheless, these works all rely on a solid secure transport layer provided by serverless platforms. We believe that our work is complementary to existing research on serverless security and access control.

Virtualization Optimization. Several research efforts have attempted to develop lightweight virtualization techniques to optimize the efficiency-security trade-off. AWS Firecracker

[1] and SEUSS [15] devised lightweight VMs (microVMs) to accelerate function initialization. For instance, AWS Firecracker [1] removes unnecessary features like BIOS, PCI, and multi-OS support from traditional VMs. On the other hand, gVisor [80] is a new security-oriented container design to guarantee strong isolation between the host OS and containers. SCONE [4] utilizes the Intel SGX trusted computing to provide a secure container mechanism. All of these designs can initialize a serverless function at the millisecond scale, but make the adverse impact of connection setup latency more significant. The approach adopted by QFaaS is fundamentally different, but is also complementary to them.

Scaling and Scheduling. The cold-start problem is a major drawback of serverless computing [34]. To achieve low cold-start latency, vigorous approaches are proposed. SAND [2] used fine-grained application sandboxing and hierarchical message bus mechanisms. FnSched [69] mitigated the resource contention between collocated functions by dynamically regulating the CPU shares. Nightcore [33] combined multiple techniques in platform design, including a fast path for internal function calls, efficient threading for I/O, *etc.* Obetz et al. [56] and Archipelago [66] proposed to use graph analysis to schedule function initialization in an efficient way. QFaaS solves the cold-start problem from a different angle of existing efforts. It can be combined with aforementioned approaches to better tame this problem.

Serverless Performance Benchmarks. Benchmark suites, such as ServerlessBench [81], SPEC-RG [76], FAASDOM [45], and PanOpticon [67], were designed for serverless platforms to characterize metrics such as communication efficiency, stateless overhead, and performance isolation in different ways. Research literature, including [28, 37, 40, 46, 78], measured the performance differences in elasticity, latency, reliability, I/O, and cost for major commercial serverless platforms such as AWS, Google, Azure, and IBM. Our evaluation design mainly draws on their work to demonstrate the benefits of QFaaS in accelerating serverless networks.

6.2 QUIC: Evolution and Extensions

QUIC was first released by Google in 2013 [19], which was informed by their experiments with the SPDY protocol [74]. This QUIC edition was later called gQUIC and brought to the IETF in 2015. Google joined the IETF team to provide a standardized protocol implementation called IETF QUIC, which has been incorporated into the Chrome browser since Oct. 2020 [20] and released as RFC 9000 [60] in May 2021. Numerous open-source efforts, including major cloud providers, have joined to provide the QUIC implementation based on the IETF standard in different programming languages, such as `quic-go` (Go) [43], `MsQuic` (C, Microsoft) [50], `mvfst` (C++, Facebook) [27], and `quiche` (Rust, Cloudflare) [22].

After its success on the wide-area Internet, such as web surfing and video streaming, QUIC has recently been extended to other broader network scenarios. Kumar et al. [38] utilized QUIC in IoT scenarios and have shown that QUIC largely benefits the connection migration for IoT devices. Thomas et al. [75] demonstrate that compared with TLS, QUIC can halve the page load time over the public satellite communication system. Research literature, such as [12, 13], integrated QUIC into the Tor network and provided empirical evaluation to show network acceleration.

Cicconetti et al. [21] conducted a preliminary evaluation of the benefits of using QUIC for end-user to FaaS Gateway connections in mobile networks (connection ❶|❽ in Figure 2 (b)). This is just another use case of QUIC on the wide-area Internet and does not consider the new challenges brought by the serverless paradigm in intra-platform connections between Gateway and function workers (❷|❼, ❸|❻, and ❹|❺). To the best of our knowledge, QFaaS is the first work to extend QUIC into the domain of serverless cloud platforms.

7 CONCLUSION

In this paper, we raise the challenge of accelerating communications while providing security in emerging serverless cloud networks. To that end, we first abstract the network communication model for serverless computing systems and then propose an extension of the QUIC protocol, called QFaaS, that provides low latency serverless function communication with improved security. We implement the QFaaS prototype on the popular OpenFaaS platform such that it requires no code modification for cloud tenants to gain network performance boosts and security benefits. QFaaS function chain library and always-on 0-RTT designs can further accelerate serverless networking. Additionally, the QFaaS design can also be easily extended to other prevalent serverless platforms. Our evaluations on synthetic serverless functions and real-world serverless applications demonstrate that QFaaS can reduce the end-user response latency by 28% (in 0-RTT mode) and 14% (in 1-RTT mode) compared to OpenFaaS using TCP+TLS. We find that the performance benefits of QFaaS linearly increase with the length of the function chain. This was also validated against several real-world serverless applications, where QFaaS obtained a maximum 50 ms reduction in latency. Overall, our findings validate that QFaaS delivers a compelling performance and security enhancement to the ecosystem of open-source serverless platforms.

ACKNOWLEDGMENTS

We thank our shepherd, Haggai Eran, and all anonymous reviewers for their insightful feedback. This project is based on work supported by the National Science Foundation (NSF) under grant CNS 2229455.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *USENIX NSDI*.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *USENIX ATC*.

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. In *Object-Oriented Programming, Systems, Languages and Applications (OOPLSA 18)*. ACM.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. SCONE: Secure linux containers with intel SGX. In *USENIX OSDI*.

[5] AWS. 2014. Lambda – Serverless Computing. https://aws.amazon.com/lambda/.

[6] AWS. 2019. *Announcing improved VPC networking for AWS Lambda functions.* https://aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/ Accessed on 2022-06-08.

[7] AWS. 2020. *Amazon Web Services: Overview of Security Processes.* https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Whitepaper.pdf Accessed on 2022-06-08.

[8] AWS. 2021. *Data protection in Amazon EC2 - encryption in transit.* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/data-protection.html#encryption-transit Accessed on 2022-06-08.

[9] AWS. 2021. *Lambda, AWS Boto3.* https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/lambda.htmll

[10] AWS. 2021. *Security Overview of AWS Lambda: An In-Depth Look at AWS Lambda Security.* https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf Accessed on 2022-06-08.

[11] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer.

[12] Lamiaa Basyoni, Aiman Erbad, Mashael Alsabah, Noora Fetais, and Mohsen Guizani. 2019. Empirical performance evaluation of QUIC protocol for Tor anonymity network. In *15th International Wireless Communications & Mobile Computing Conference (IWCMC 19)*. IEEE.

[13] Lamiaa Basyoni, Aiman Erbad, Mashael Alsabah, Noora Fetais, Amr Mohamed, and Mohsen Guizani. 2021. QuicTor: Enhancing Tor for Real-Time Communication Using QUIC Transport Protocol. *IEEE Access* 9 (2021), 28769–28784.

[14] M. Bishop. 2022. *HTTP/3.* RFC 9114. IETF. https://www.rfc-editor.org/rfc/rfc9114.txt

[15] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys 20)*.

[16] Xudong Cao, Shangru Zhao, and Yuqing Zhang. 2019. 0-RTT Attack and Defense of QUIC Protocol. In *2019 IEEE Globecom Workshops (GC Wkshps 19)*. IEEE.

[17] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM* 62, 12 (2019), 44–54.

[18] Shan Chen, Samuel Jero, Matthew Jagielski, Alexandra Boldyreva, and Cristina Nita-Rotaru. 2019. Secure communication channel establishment: TLS 1.3 (over TCP fast open) vs. QUIC. In *European Symposium on Research in Computer Security (ESORICS 2019)*. Springer.

[19] Chromium Blog. 2013. *Experimenting with QUIC.* https://blog.chromium.org/2013/06/experimenting-with-quic.html Accessed on 2022-06-08.

[20] Chromium Blog. 2020. *Chrome is deploying HTTP/3 and IETF QUIC.* https://blog.chromium.org/2020/10/chrome-is-deploying-http3-and-ietf-quic.html Accessed on 2022-06-08.

[21] Claudio Cicconetti, Leonardo Lossi, Enzo Mingozzi, and Andrea Passarella. 2021. A Preliminary Evaluation of QUIC for Mobile Serverless Edge Applications. In *22nd IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*.

[22] Cloudflare. 2018. *quiche: Savoury implementation of the QUIC transport protocol and HTTP/3.* https://github.com/cloudflare/quiche

[23] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *Proceedings of The Web Conference*.

[24] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. 2019. Pluginizing quic. In *ACM SIGCOMM*.

[25] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3.* RFC 8446. IETF.

[26] Lars Eggert. 2020. Towards securing the internet of things with quic. In *Workshop on Decentralized IoT Systems and Security (DISS 20)*.

[27] Facebook. 2019. *mvfst: An implementation of the QUIC transport protocol.* https://github.com/facebookincubator/mvfst

[28] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4792.

[29] Google. 2017. *Encryption in Transit in Google Cloud.* https://cloud.google.com/security/encryption-in-transit Accessed on 2022-06-08.

[30] Google. 2021. *Google Cloud Security Whitepapers.* https://cloud.google.com/security/overview/whitepaper Accessed on 2022-06-08.

[31] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 2017. 0-RTT key exchange with full forward secrecy. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt 17)*. Springer.

[32] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. 2010. Early observations on the performance of Windows Azure. In *ACM HPDC*.

[33] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ACM ASPLOS*.

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

[35] Kenneth Reitz. 2011. Requests: HTTP for Humans. https://docs.python-requests.org/en/master/. https://docs.python-requests.org/en/master/

[36] Hannah Kuchler. 2015. *Hackers find suppliers are an easy way to target companies.* Accessed on 2022-06-08.

[37] Jörn Kuhlenkamp, Sebastian Werner, Maria C Borges, Dominik Ernst, and Daniel Wenzel. 2020. Benchmarking elasticity of FaaS platforms as a foundation for objective-driven design of serverless applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC 20)*.

[38] Puneet Kumar and Behnam Dezfouli. 2019. Implementation and analysis of QUIC for MQTT. *Computer Networks* 150 (2019), 28–45.

[39] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*.

[40] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD 18)*. IEEE.

[41] Shuhao Liu, Hong Xu, and Zhiping Cai. 2013. Low latency datacenter networking: A short survey. *arXiv preprint 1312.3455* (2013).

[42] OpenFaaS Ltd. 2016. OpenFaaS: Serverless Functions, Made Simple. https://www.openfaas.com/.

[43] Lucas Clemente, et. al. 2016. *quic-go: A QUIC implementation in pure Go.* https://github.com/lucas-clemente/quic-go

[44] M. Thomson, S. Turner. 2021. *Using TLS to Secure QUIC.* RFC 9001. IETF.

[45] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS 20)*.

[46] Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima. 2017. Benchmarking heterogeneous cloud functions. In *European Conference on Parallel Processing (Euro-Par 17)*. Springer.

[47] Markets and Markets. 2020. *Serverless Architecture Market by Service Type (Automation and Integration, Monitoring, API Management, Security, Analytics, and Design and Consulting), Deployment Model, Organization Size, Vertical, and Region - Global Forecast to 2025.* https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.html Accessed on 2022-06-08.

[48] Stephen Paul Marsh. 1994. Formalising trust as a computational concept. *University of Stirling* (1994).

[49] Matt Adorjan. 2021. *AWS Latency Monitoring.* https://www.cloudping.co/grid

[50] Microsoft. 2019. *MsQuic: Cross-platform, C implementation of the IETF QUIC protocol.* https://github.com/microsoft/msquic

[51] Microsoft. 2021. *Azure encryption overview.* https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-overview Accessed on 2022-06-08.

[52] Microsoft. 2021. *Azure security baseline for Azure Functions.* https://docs.microsoft.com/en-us/security/benchmark/azure/baselines/functions-security-baseline Accessed on 2022-06-08.

[53] Nordstrom. 2017. *Towards a serverless event-sourced Nordstrom.* https://youtu.be/WcCErxLKR7g

[54] Nordstrom Technology. 2019. *Hello, Retail!* https://github.com/Nordstrom/hello-retail

[55] Matthew Obetz, Anirban Das, Timothy Castiglia, Stacy Patterson, and Ana Milanova. 2020. Formalizing event-driven behavior of serverless applications. In *European Conference on Service-Oriented and Cloud Computing (ESOCC 20)*. Springer.

[56] Matthew Obetz, Stacy Patterson, and Ana Milanova. 2019. Static Call Graph Construction in AWS Lambda Serverless Applications. In *USENIX HotCloud*.

[57] OpenFaaS Ltd. 2021. *GitHub: openfaas/faas.* https://github.com/openfaas/faas

[58] OpenFaaS Ltd. 2022. *OpenFaaS Function Store.* https://github.com/openfaas/store

[59] Diana Popescu, Noa Zilberman, and Andrew Moore. 2017. Characterizing the impact of network latency on cloud-based applications' performance. *Computer Laboratory technical reports* (2017).

[60] QUIC Working Group. 2021. *QUIC: A UDP-Based Multiplexed and Secure Transport.* RFC 9000. IETF.

[61] R. Fielding, J. Gettys , J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. IETF.

[62] Raphael Satter, Christopher Bing, Joseph Menn. 2020. *Hackers used SolarWinds' dominance against it in sprawling spy campaign.* https://www.reuters.com/article/global-cyber-solarwinds/hackers-at-center-of-sprawling-spy-campaign-turned-solarwinds-dominance-against-it-idUSKBN28P2N8 Accessed on 2022-06-08.

[63] Scott W Rose, Oliver Borchert, Stuart Mitchell, and Sean Connelly. 2020. Zero trust architecture. *Special Publication, National Institute of Standards and Technology (NIST SP), Gaithersburg, MD* (2020).

[64] Arnav Sankaran, Pubali Datta, and Adam Bates. 2020. Workflow Integration Alleviates Identity and Access Management in Serverless Computing. In *Annual Computer Security Applications Conference (ACSAC 20)*.

[65] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX ATC*.

[66] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2019. Archipelago: A scalable low-latency serverless platform. *arXiv preprint 1911.09849* (2019).

[67] Nikhila Somu, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Panopticon: A comprehensive benchmarking tool for serverless applications. In *2020 International Conference on Communication Systems & NETworkS (COMSNETS 20)*. IEEE.

[68] Stefan Prodan. 2022. *GitHub: stefanprodan/openfaas-certinfo.* https://github.com/stefanprodan/openfaas-certinfo

[69] Amoghvarsha Suresh and Anshul Gandhi. 2019. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*.

[70] Erik Sy, Tobias Mueller, Christian Burkert, Hannes Federrath, and Mathias Fischer. 2020. Enhanced Performance and Privacy for TLS over TCP Fast Open. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 271–287.

[71] Lizhuang Tan, Wei Su, Yanwen Liu, Xiaochuan Gao, and Wei Zhang. 2021. DCQUIC: Flexible and Reliable Software-defined Data Center Transport. In *IEEE INFOCOM Workshops*. IEEE.

[72] Tarun Mangukiya. 2022. *GitHub: tarunmangukiya/openfaas-functions.* https://github.com/tarunmangukiya/openfaas-functions

[73] The Apache Software Foundation. 2016. OpenWhisk, Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[74] The Chromium Projects. 2010. *SPDY: An experimental protocol for a faster web.* http://www.chromium.org/spdy/spdy-whitepaper

[75] Ludovic Thomas, Emmanuel Dubois, Nicolas Kuhn, and Emmanuel Lochin. 2019. Google QUIC performance over a public SATCOM access. *International Journal of Satellite Communications and Networking* 37, 6 (2019), 601–611.

[76] Erwin Van Eyk, Joel Scheuner, Simon Eismann, Cristina L Abad, and Alexandru Iosup. 2020. Beyond microbenchmarks: The spec-rg vision for a comprehensive serverless benchmark. In *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE 20)*.

[77] W3Techs. 2021. *Usage statistics of QUIC for websites.* https://w3techs.com/technologies/details/ce-quic

[78] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *USENIX ATC*.

[79] Y. Chen, J. Chu, S. Radhakrishnan, A. Jain. 2014. *TCP Fast Open.* RFC 7412. IETF.

[80] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2019. The true cost of containing: A gVisor case study. In *USENIX HotCloud*.

[81] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 20)*.