National College of Ireland

Improve Cold Start Latency for Stateful Serverless Functions Using Pooled Resource Injection

MSc Research Project

MSCCLOUD

Tony Doolin

Student ID: 23420855

School of Computing

National College of Ireland

Supervisor: Sai Emani

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Tony Doolin
Student ID:	23420855
Programme:	MSCCLOUD_A **Year:** 2024
Module:	Research Project
Supervisor:	Sai Emani
Submission Due Date:	14 September 2025
Project Title:	Improve Cold Start Latency for Stateful Serverless Functions Using Pooled Resource Injection
Word Count:	~ 7000 **Page Count:** 21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: 

Date: 14 September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Improve Cold Start Time for Stateful Serverless Functions Using Pooled Resource Injection

Tony Doolin

23420855

Abstract

Serverless computing allows applications to be easily deployed and takes away a lot of the complexity of scalability and maintainability. However cold start latency issues limit its adoption for stateful applications like Online Transaction Processing (OLTP) systems. This problem is compounded by the additional cost of establishing network connections to backed database systems for each serverless function invocation, as these functions cannot maintain state between invocations. This research investigates a novel approach to reducing cold start latency by injecting pre-established TCP connections into serverless containers during startup. A Connection Pool Manager was implemented to maintain active network connections and distribute them to serverless functions via Inter-Process Communication using file descriptor passing. Experimental evaluation using a serverless function deployed on a Docker container in the cloud compared direct connection establishment against connection injection across 1000 test iterations for varying connection establishment speeds. Results show that connection injection achieves connection establishment times 3.6 times faster than direct connections (260µs vs 951µs average). It also has reduced variance and improved predictability. Overall container lifecycle times improved by 4.4%, with 99th percentile response times decreasing from 514ms to 439ms. However, the research also encountered practical limitations regarding TLS connection injection and Kubernetes network namespace restrictions which limits its use in the real world. Notwithstanding this, this work provides valuable insights for improving serverless performance in stateful use cases and offers a possible approach for migrating OLTP workloads to serverless platforms.

1 Introduction

1.1 Research Background

In 2008, Google created the serverless programming paradigm when it introduced Google App Engine (McDonald, 2008). Since then, serverless programming has become a widely adopted means of developing and deploying software in the cloud and Function as a Service (FaaS) platforms are now offered by all major cloud providers. They offer significant advantages over more traditional IaaS and PaaS deployment models due to their ease of development, scalability and simplicity of deployment (Eismann *et al.*, 2021) . This is coupled with a pay-as-you-use cost model which makes it easy to start small and grow as required.

By contrast, financial Online Transaction Processing (OLTP) is an area that has historically been implemented using powerful mainframe systems tightly coupled to backend relational databases deployed in private data centres. This has allowed these systems to provide strong reliability, consistency and response time SLAs which are crucial for this type of application. However, there are a number of issues with these systems including a high cost of operation, complexity of maintenance and difficulty in dynamically scaling the platform. (Govindaraj, 2024).

These challenges are largely addressed by cloud platforms, particularly serverless, as they are dynamically scalable, easy to develop and enhance, and relatively simple to deploy (Jonas *et al.*, 2019). However, there are a number of reasons why implementing OLTP systems on serverless platforms is still not widely adopted.

Firstly, serverless platforms do not manage transactions and persistent state very well. In general, the use cases implemented by serverless platforms are stateless (Joosen *et al.*, 2023) ie the function does not modify or interact with persistent storage or databases. This is generally by design and introducing state to a serverless function can cause issues with performance and resource management (Mampage, Karunasekera and Buyya, 2022).

The second reason is because serverless systems struggle to provide consistent SLAs, particularly for systems which require a short response time. This is due to a number of factors, including the dynamic management of resources, the often bursty nature of traffic, and that intermediate queues and gateways are generally used to transfer data to and from the processing function (Nguyen *et al.*, 2019) . This is further exacerbated by the Cold Start Latency problem that serverless systems suffer from

Cold Start Latency is the delay that occurs when a serverless function is invoked after it has been idle for a period or there is a scaling event. In these cases, the FaaS platform does not have enough resources to execute the function and must provision additional resources in order to execute the function. These resources are usually a container or VM environment and can cause significant variance in how long it takes to execute a function (Golec *et al.*, 2023) .

The issues described above apply generally to serverless platforms. However, there is a further overhead which must be considered for OLTP systems, which is the cost of connecting to a downstream RDBMS system. In traditional processing systems, resources such as database connections can be provisioned during the system startup and allocated to processing threads as the require them. This pattern does not work for serverless functions because they are invoked in isolation without any global context or state, and so all resources they require can only be provisioned once the function has been invoked.

The problems of addressing the Cold Start Latency issue in serverless computing has been widely addressed. Solutions to the problem fall into 2 broad categories: 1) Reducing the frequency of cold start events and 2) Reducing the time it takes to start a new container (Golec *et al.*, 2023).

Reducing the frequency of cold start events generally involves keeping containers warm (i.e. not shutting them down when they are idle) or predicting when the containers may be required

and pre-warming them using Machine Learning techniques (Bermbach, Karakaya and Buchholz, 2020). While this approach has yielded success, it comes with the overhead of having to use additional resources and cost to achieve the solution, reducing the impact of one of the main reasons to adopt serverless computing

Reducing the startup time has also been widely adopted, and the approach usually involves checkpointing a container directly after it has been started but before it has started executing the function. so that it can be started faster on subsequent invocations. This technique uses Checkpoint/Restore in Userspace (CRIU) and is often referred to as "pre-baking" the container (Fireman *et al.*, 2024). However, this technique also has limitations, especially in the area of network connection restoration. In order to restore the connection correctly, the image needs to be restored from the same host it was created which limits the usage of the image on a system which has multiple hosts,

1.2 Problem Statement

Serverless systems are primarily designed to implement stateless functions and as a result do not prioritise addressing drawbacks that would enable stateful use cases such as OLTP processing to be migrated from their traditional mainframe platforms to serverless cloud platforms. Specifically, cold start latencies mean that committing to specific SLAs for processing response times is hard on serverless platforms, and this is one of the key requirements of an OLTP system. The problem of cold start latencies is further magnified by the specific need for an OLTP serverless function to establish network connections to a database each time it is invoked in order to manage state associated with the transaction.

1.3 Research Question

To what extent does injecting an active network connection into a serverless function improve the startup time and overall processing time of a function which needs to make a network connection?

1.4 Research Objective

In order to answer the research question, I will review the current state of research around cold start latency, specifically for stateful use cases. I will then design and implement a means of injecting network connections into a serverless function during container startup. Finally, I will measure the reduction in startup latency and end-to-end processing time for a container to see what performance improvements have been made

1.5 Research Contributions

My research will provide the following contributions to the area of Cold Start research:

- Propose a novel way of reducing the cold start latency for stateful serverless use cases
- Show what performance improvements can be made by this novel approach

- Compare this to the standard way of directly connecting toa database as part of the function processing.

2 Related Work

2.1 Serverless Platforms

The use of serverless platforms have grown enormously since their introduction in 2008 by Google They remained a relatively niche cloud technology till 2014, when Amazon introduced its Lambda Function-as-a-Service (FaaS) platform at which point it started gaining more widespread adoption.

The popularity of serverless and FaaS has grown due to a number of key differentiators from traditional IaaS and PaaS platforms. It abstracts resource management away from the developer, simplifying the development process. Dynamic scaling is also greatly simplified, resulting in a generally simpler deployment and operational model. Finally, it's pay-as-you-go cost model means organizations can start small with their adoption of serverless technologies without having to invest large sums in infrastructure (Jonas *et al.*, 2019).

The use cases for serverless span a wide range of industries from Healthcare, Transport Systems, AI & ML Enterprise Data Intelligence to IOT-based applications (Golec *et al.*, 2023).

However, in spite of these significant advantages, there remain a number of challenges with serverless computing. Cold Start Latency is where a new container needs to be provisioned in order to execute a function that has not been executed for a period. This issue results in un-predictable execution times for the functions, meaning it can be hard to commit to Service Level Agreements (SLAs) for a FaaS platform (Lannurien Vincent and D'Orazio, 2023).

Managing state is also a challenge for FaaS systems. Serverless platforms are inherently designed to be stateless. They execute in an ephemeral environment, working on data and events that have been passed as input via an event or API call. This mechanism works well for may use cases and is one of the reasons why serverless are simple to operate and scale. However, there are many use cases which do require state, including OLTP systems, and using state for these use cases is a significant challenge.

2.2 Cold Start Latency

There has been much research into the challenge of Cold Start Latency in Serverless computing. The approaches break down into two distinct categories.

2.2.1 Reducing the Frequency of Cold Starts

This approach can be a relatively simple fix to the Cold Start Latency problem – Simply keep the containers alive when they are idle, and this means they do not have to restart. This feature is provided by the major FaaS providers, including AWS, Azure and GCP through the capability to keep a set of pre-warmed resources permanently available even if the system is idle. However, these features hide a number of issues that go against some of the main

advantages of serverless computing. The first of these is that resources are only used on demand. Having pre-provisioned resources available means that these resources are not available for other purposes, potentially limiting dynamic scaling for other parts of the system. Secondly, it also increases the cost of the deployment, limiting another of the advantages of serverless systems which is that you only pay for the resources you require.

There are other more sophisticated approaches to pre-warming containers on FaaS platforms. Machine Learning such as Long Short-term Memory (LSTM) analysis can be done on the FaaS platform in order to predict when demand will increase, and containers are pre-started before the demand ramps up (Xu *et al.*, 2019). Other systems such as ATOM (Golec *et al.*, 2024) use Deep Learning methods to analyze the patterns of requests coming to the system and build a Time Series model to predict future traffic patterns. While these systems have been shown to successfully reduce the number of cold starts in systems, they require their own resources to perform the analysis, and also introduce significant complexity to the system, so they can actually increase the overall cost of systems, especially for smaller systems or IoT deployments.

Function Fusion is another approach to eliminating the need to start containers. This works by analyzing FaaS workflows and identifying separate functions which can be combined into a single function (Schirmer *et al.*, 2024). Reducing the number of functions in the system will inevitably lead to less container starts. The downside to this approach is two-fold. Firstly, depending on how the system is designed, there may not be points where function fusion makes sense so this approach may not be applicable to all FaaS systems. The second issue is that it encourages building monolithic systems, or at least systems which may not be appropriately de-composed, leading to potentially poor design decisions.

2.2.2 Reducing the Cold Start Latency

The other approach to tackling the Cold Start Latency problem is to reduce the actual time it takes to start the container. The main mechanism for achieving this is by taking a snapshot of a container image directly after it has started but before it has started processing. This allows for fast restoration of the image the next time it starts. This capability is implemented using the Linux Checkpoint/Restore in Userspace utility (*CRIU*, 2025). This approach is taken by Catalyzer (Du *et al.*, 2020) and through a mechanism called “pre-baking” (Silva, Fireman and Pereira, 2020). Pre-baking picks a static point during the container startup process and saves an image of the container for restoration when the container is required again. This approach was further optimized by Pronghorn (Kohli *et al.*, 2024) , which dynamically updated the point of snapshotting based on a performance feedback loop.

However, using CRIU for checkpointing of open network connections has some restrictions. Connections can only be restored if the image was created on the same host as the network connection (*Live migration - CRIU*, 2023). So for network connection restoration to work correctly, either an image of each container must be created for each host, or there is only a 1/N chance of a network connection being restored for N hosts in a FaaS deployment.

Networking has previously been identified as a significant contributor to Cold Start Latency. The Pause Container Pool Manager (Mohan *et al.*, no date) proposes a way of pre-provisioning network resources including namespaces to speed up the start process of Docker containers. However, this work focusses on the internal network infrastructure required by the container, it does not address optimizations for external network connections.

2.3 Stateful Serverless platforms

In spite of the fact that serverless platforms are best suited to stateless use case, there are serverless systems which are specifically focussed on managing stateful use cases. These systems generally focus on incorporating transactional behaviour into the serverless workflow by adding the capability for serializable transactions, two-phase commit and Saga patterns (de Heus *et al.*, 2022). Systems such as Beldi (Xu *et al.*, 2019) provide a framework for implementing consistency and isolation semantics on top of serverless platforms such as AWS Lambda functions.

Other systems such as Crucial (Barcelona-Pons *et al.*, 2022) enable sharing state across multiple instances of a function by using distributed shared objects. This is primarily for ML use cases such as k-means clustering algorithms.

2.4 Critical Analysis

As can be seen from the previous analysis, there is a lot of research on both Stateful Serverless platforms, and Cold Start Latency. However, there appears to be little or no overlap between these two fields of research.

In the case of Cold Start Latency, the pre-baking system used 3 use cases – No-Op, A Markdown renderer and an image re-sizer. Pronghorn had a wider set of test cases, including functions such Breadth-first search on random graphs, video processing and HTML generation. However, in all cases, the functions were stateless and did not consider either persistent storage or network connectivity.

And for the Stateful serverless systems, the focus is understandably on maintaining transactional integrity and state rather than addressing cold start issues. The majority of systems do not even mention cold start as an issue. The Crucial paper does mention it but only calls it out as a test artefact and does not address any way to overcome the issue.

So clearly there is a gap in the research in this area which can be tackled with further analysis and research. Specifically, optimizing network connectivity is a unique crossover point between Stateful Serverless platforms and Cold Start Latency which has not been adequately addressed by current research, and would greatly contribute to enabling systems such as OLTP meet both functional and performance requirements.

3 Research Methodology

3.1 Research Problem

The objective of this research is to determine by how much injecting an active network connection into a serverless container on startup will reduce the overall latency of a transaction that it is processing.

3.2 Development Framework

In order to measure the impact of passing an active network connection, I require a system which has the capability to maintain a pool of open connections per host, and then a means of a container requesting one of these connections on startup and using it during function execution. None of the existing FaaS platforms have this feature, so I needed to build the capability into an existing FaaS platform. I chose OpenFaaS, as it is a widely adopted open-source FaaS platform which has ongoing maintenance and documentation.

The fundamental element of this system requires a component which maintains a pool of open connections to a server on one side, and an interface to distribute these open connections to the container on the other, called a Connection Pool Manager.

On startup, the container connects to a Connection Pool Manager to retrieve an open connection on container startup and then utilizes this connection during its processing.

To distribute the open connections, the connection pool server passes the file descriptor for the connection over an Inter-Process Communication (IPC) interface. File descriptors are a fundamental feature of the Linux operating system, providing a means for the OS and applications to manage file and network connections. When a file or network connection is opened by an application, a file descriptor is used to access the network connection. File descriptors are OS-level, system-wide resources and they can be passed from application to application on the same server.

3.2.1 Full OpenFaaS Deployment

The initial plan was to deploy a full OpenFaaS system. This involved deploying a Kubernetes cluster of 3 systems, a master node and 2 worker nodes. After deploying Kubernetes, I installed OpenFaaS on top of it, using the Arkade deployment mechanism (*OpenFaaS CE - OpenFaaS*, 2024).

However, a significant issue arose in that it was not possible to pass open file descriptors to containers deployed in Kubernetes. This was because Kubernetes uses dedicated network namespaces for the pods it deploys. One of the features of network namespaces is that it isolates networking resources and so it is not possible to open a network connection in one namespace and transfer it to another (*Services, Load Balancing, and Networking | Kubernetes*, 2024) (*network_namespaces(7) - Linux manual page*, 2024), so I needed to take a different approach.

3.2.2 Simplified faasd Deployment

OpenFaaS also provides a simplified means of deployment, using faasd (*openfaas/faasd: Lightweight and portable version of OpenFaaS*, 2025), which is a standalone daemon which provides all of the basic capabilities of OpenFaaS but does not deploy using Kubernetes and crucially uses the host network namespace by default which means that it does not have the same restrictions with network namespaces as Kubernetes. However, this system also has a blocking limitation in that one of the design choices of the OpenFaas team was to block sharing of host resources such as files with the OpenFaaS containers (*Proposal: Adding mount options to functions. · Issue #320 · openfaas/faas*, 2022). The design choice was founded on the principle that the serverless function should not have any direct dependency on any host resources. While this is a legitimate objective, it prevented me from using socket-based IPC to transfer network connections between the Connection Pool Manager on the host and the container function as this mechanism requires the sharing of a file handle between the host and the container.

3.2.3 Standalone Container deployment

Based on these limitations, I designed a simpler solution that does not require the use of Open Faas but just uses a simpler Docker container deployed using containerd. Both Open FaaS, Kubernetes and OpenWhisk use containerd as their container engine under the hood, so this approach mimics these environments closely, ensuring that any results gathered using this approach can be applied to these environments.

3.2.4 TCP and TLS connections

The final issue I encountered while designing the research methodology was the usage of Transport Layer Security (TLS). One of the premises of my research question was an assumption that connections would be encrypted using TLS, as is standard in enterprise environments where zero-trust interfaces are the norm. In general, using TLS significantly increases the cost of establishing a connection, and thus would significantly increase the benefit of using an already-established connection. As mentioned in the literature review, previous research has shown that it can take on average 50ms to establish a TLS connection. To validate this, I ran a test in an AWS cloud connecting 2 EC2 hosts to each other using TLS 1.3. I measured the average and 95th percentile time for connection establishment for TCP and TLS 1.3. As can be seen from Table 1, establishing an encrypted connection using TLS is an order of magnitude faster than the previous research had indicated at 5ms instead of 50ms, but it still takes almost 10 times longer than establishing a TCP connection:

	TCP	**TLS 1.3**
Average (µs)	673.2035	5995.706
95th Percentile(µs)	763.1063	6491.828

Table 1

This indicates that there is potential for a significant benefit to using connection injection when combined with a requirement to use TLS connections.

However, a blocking issue arose during implementation which prevented the use of TLS connections. As part of its connection establishment, TLS maintains state in the application user space to track information such as the cipher suite negotiated during connection establishment and sequence numbers used as part of the encryption process for each packet(Rescorla, 2018). This information is not transferred across with the File Descriptor for the connection, so while the client can send data to the server over the TLS connection, the server cannot de-crypt the data. Because of these issues I cannot use TLS connections as part of my experiment. SO to work around this, I added the capability to my system to add an optional delay to the TCP handshake process. This allowed me to simulate different connection times and see how this would impact the overall lifecycle of the container.

3.3 Use cases

There are 2 use cases that will be measured:

The first is a container function starting cold, connecting over TCP directly to a server on a different host, sending data, and receiving a response.

The second is a container function connecting to a Connection Pool Manager on start-up using a Socket IPC connection and receiving an open TCP network connection to the server on the other host. The function then uses this TCP connection to send and receive data to the remote server.

3.4 Data gathering and measurement

The system will measure 2 significant points of data. Firstly, it will measure the time it takes the container function to establish a connection using both of the methods above. This measurement will not include any subsequent data transfer or processing done by the container function, as this is not relevant to answering the research question. Secondly it will measure the overall container lifetime while executing both use cases. These data points will give a means to measure whether connection establishment is faster using the traditional method of direct connection establishment or whether passing the open connection across an IPC socket is faster.

In order to establish a clear pattern of behaviour across each use case and make sure that I have a statistically significant sample size, I will run 1000 iterations of each test and use that data for analysis

3.5 Research Resources

I will execute my experiment on an AWS cloud system, using 2 EC2 VM instances connected to the same Virtual Private Network. One EC2 instance will act as a TCP server, the other will act as the container host and Connection Pool Manager. The container host will need to be capable of running 5 simultaneous containers. Initial testing has shown that the t2 and t3.micro

size is sufficient for this purpose – I will use t3.micro where possible, but some locations do not have t3.micro , so I will use t2.micro in in those cases.

4 Design Specification

4.1 System Architecture Overview

The overall system consists of 3 separate components, as shown in Figure 1.

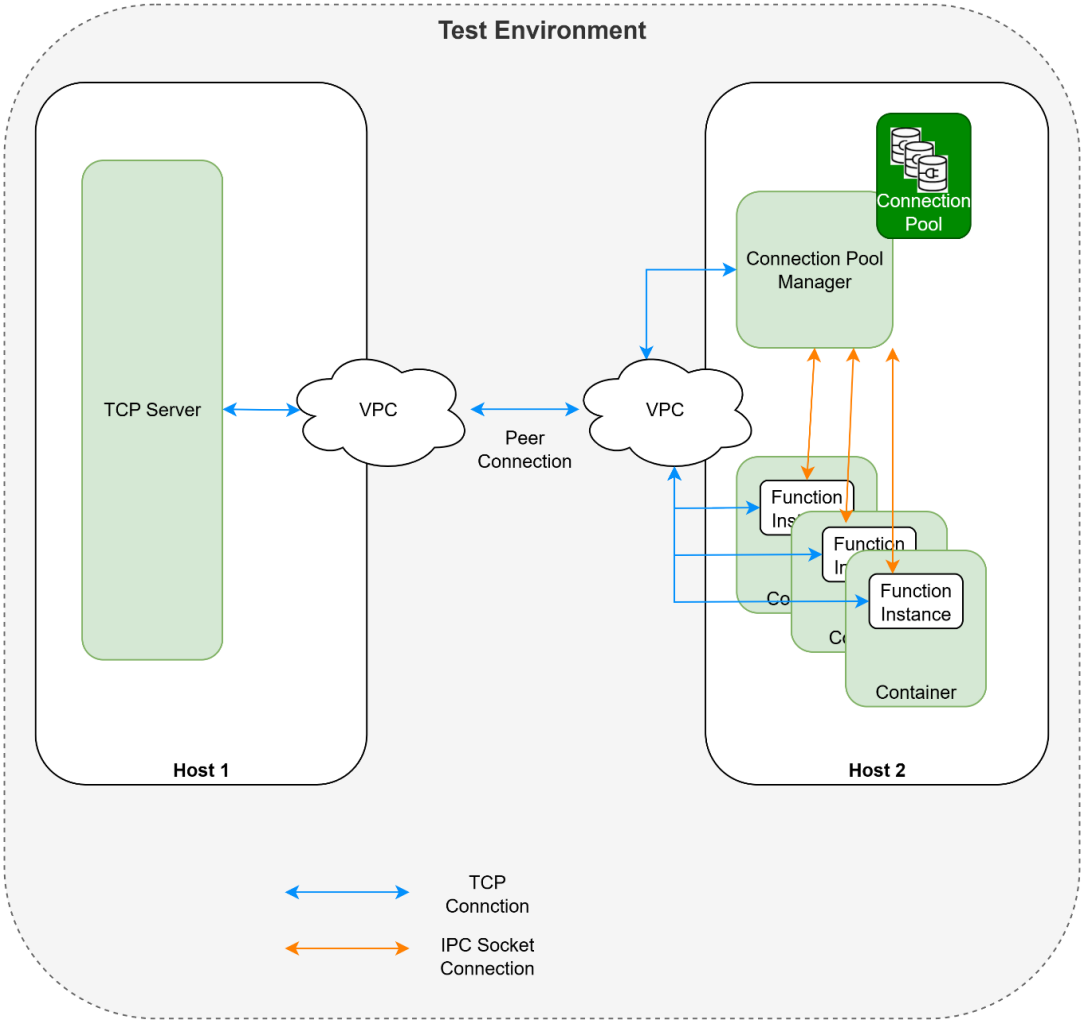

Figure 1

TCP Server is a process that will accept TCP connections from a client. Any data that is send to it by a client will be mirrored back to the client.

Connection Pool Manager is a process that performs 2 functions:

1. Establish a configurable number of connections with the TCP server and hold them open in a connection pool for future use

2. Open an IPC socket connection and send an open TCP connection to any clients that connect to it across the socket

Function Instance is a process deployed in a container which sends requests to the TCP server, and operates in 2 modes:

1. Direct connection where the function instance establishes a connection directly with the TCP server
2. Using a connection from the Connection Pool Manager to connect to the TCP server

4.2 System Deployment

The system is deployed on Ubuntu 22.04 virtual machines on Amazon AWS platform. I used 2 EC2 t3.micro instances as significant CPU resources were not required

I also used containerd, Docker and Docker Compose to host the container infrastructure. Kubernetes would have been preferable, but because of the network namespace issues outlined in the Research Methodology, this was not possible

4.3 Performance considerations

The key performance consideration for the system is its capacity to establish many network connections for the connection pool. I need to capability to establish 1000 simultaneous TCP connections, and a number of simultaneous socket IPC connections. The user open file limit may come under pressure, as it's default on AWS Ubuntu images is 1024, so this value needs to be adjusted to above that in order to make sure there is no risk that the user will breach the limit.

The other consideration is how many simultaneous containers to start at one go. I initially went with 5 simultaneous container starts at once, as while I want to mimic real-world scenarios, log-jamming the TCP stack on the TCP server will artificially skew the TCP connection time to an un-realistically high number. However, when I ran a number of tests, I observed that the docker compose scaling feature initiated the start sequence in parallel but then effectively serialised the actual start-up of the containers. This resulted in a cumulatively increasing lifecycle time for each of the 5 containers, and then a reset back for the next 5 containers, as can be seen in Figure 2

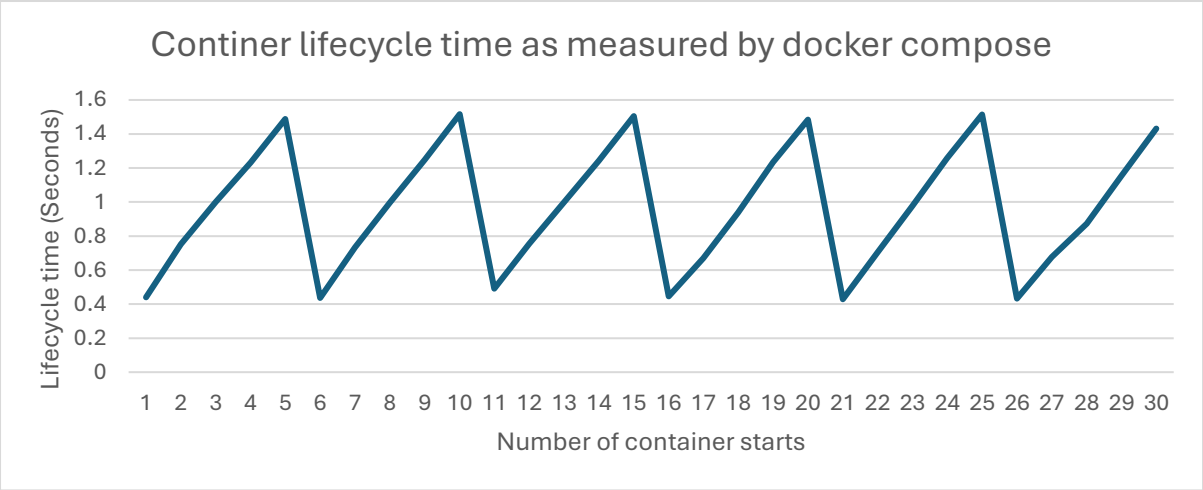

Figure 2

In order to address this, I reverted to starting the containers serially so that I would be able to accurately measure the container lifecycle time.

Finally, TCP connection establishment times can vary significantly between environments due to factors such as network speed, physical distance between servers, and number of network hops required. So in order to see how variance in the time taken to establish TCP connections could impact on the container lifecycle time, I added the capability to delay the TCP connection negotiation so that I could run experiments to measure how this variance impacts the container lifecycle time.

5 Implementation

All components were implemented using Python 3. I chose this as it is one of the more popular languages for implementing serverless functions and is representative of what is used and supported by all the major serverless platforms (Datadog, 2023). Although Python is a scripting language, this does not have a significant performance impact on TCP connection establishment and IPC communication, as the majority of the work involved in both activities is performed by the kernel.

5.1 Connection Pool Manager

The fundamental element of the test system is a component which maintains a pool of open connections to a server on one side, and an interface to distribute these open connections to the container on the other.

On start-up the Connection Pool Manager builds a pool of 1000 connections to the TCP server. Once it has completed this, it opens an IPC socket interface in order to enable the container function to access the pool of open connections.

To distribute the open connections, the container function connects to the Connection Pool Manager over the IPC socket. When a connection is accepted by the CPM, it retrieves a TCP

connection from the pool and passes the FD for the connection across the IPC socket to the container function.

5.2 Serverless Function Instance

The serverless function is quite straight-forward. It can operate in 2 modes, direct mode and socket mode. In direct mode the function connects directly to the TCP server and sends a request which is echoed back by the TCP server.

In socket mode, it connects to the IPC socket opened by the Connection Pool Manager and receives an open network file descriptor. It then uses this connection to send a request to the TCP server and receive a response.

In order to measure the time taken to make the direct and socket connections, the serverless function logs records a timestamp before it makes the connection, and another timestamp when the connection is established, or received from the Connection Pool Manager in socket mode. See Figure 1 for a diagram of where the timestamps are recorded

Note that the time to send and receive the data to the TCP server is not included in the measurement, it is just a measurement of the time it takes to establish the connection.

Note also that the container lifecycle is only for the duration of the function. Essentially, every function execution is a cold start, which enables me to measure the impact of the connection establishment approaches more efficiently.

5.3 TCP Server Implementation

The TCP server is just a simple echo server which listens on a socket, accepts incoming connections and wait for data to be sent to it. Any data that is sent is echoed back to the client. It has no awareness of whether the connection came directly from the container function or the Connection Pool Manager and does not treat the connections any differently.

5.3.1 TCP Server delay

In order to enable the system to vary the TCP connection time across experiments, I introduced a delay to the TCP connection flow. This occurs after the client has sent the SYN message and before the server responds with the SYN-ACK message, as shown in Figure 3. This is implemented using the OS tool tc (*tc(8) - Linux manual page*, 2025) and is wrapped in a script in order to simplify its usage

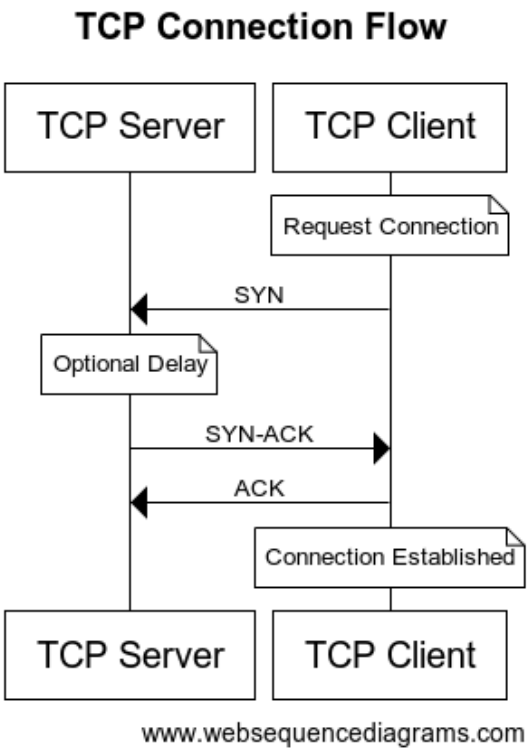

Figure 3

5.4 System Workflow & Interactions

Figure 4 shows the workflow diagram describing the interactions between each component for a single execution of the test. The lifecycle of the TCP server and Connection Pool Manager is long-lived, whereas the serverless container starts and stops repeatedly.

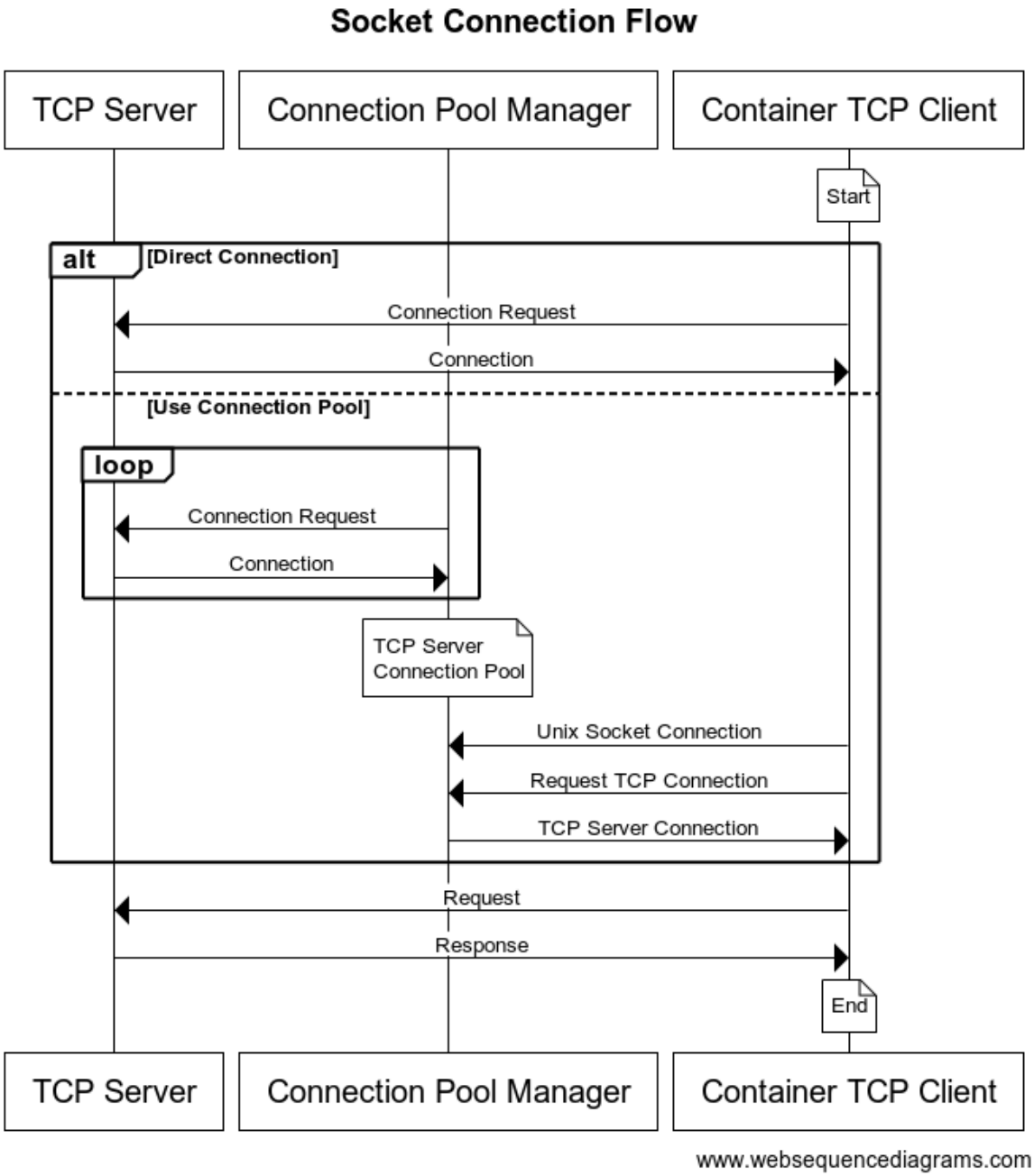

Figure 4

5.5 Deployment Environment

I used Ubuntu 22.04 LTS to run my VMs, deployed on AWS EC2 platform

5.6 Performance Measurement

I collected 2 types of measurement. The first was the time to establish the connection to the TCP server from the container function. This was captured as a log output from the container.

The second was the overall lifecycle time of the container from creation to deletion. This was captured from the docker compose logs which are automatically logged as part of the docker compose framework.

6 Evaluation

6.1 Connection time comparison

For the direct connection test, the connection to the TCP server took an average of 951µs across the 1000 executions of the serverless function. For the socket connection the connection took an average of 260 µs

This means that connecting using the Connection Pool Manager is 3.6 time faster than establishing a direct connection

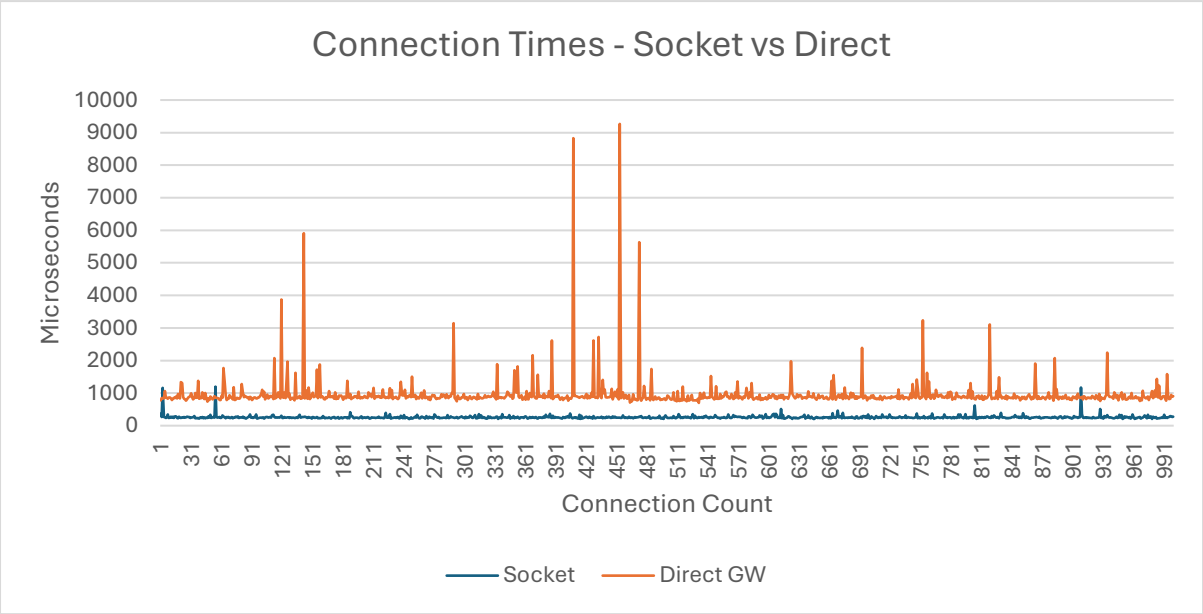

Figure 5

As can be seen in the graph in Figure 5, the other observation is that there is a lot more variance and jitter in the time taken to establish a direct connection. The 95th percentile of the direct connection is 1283µs, compared to 333µs for the Connection Pool Manager approach. This means that there is a more predictable latency associated with using socket connection pooling, which is a significant advantage when trying to commit to specific SLAs.

6.2 Container lifecycle comparison

In terms of how this affects the overall lifecycle of the container, the average lifecycle time for the socket connection tests was 338ms. For the direct connection, it is 353ms. So, using the socket connection approach gives a container lifecycle improvement 4.4%. While this is not a

large margin, it can be directly attributed to the delta in the connection times between the direct and the socket connection times.

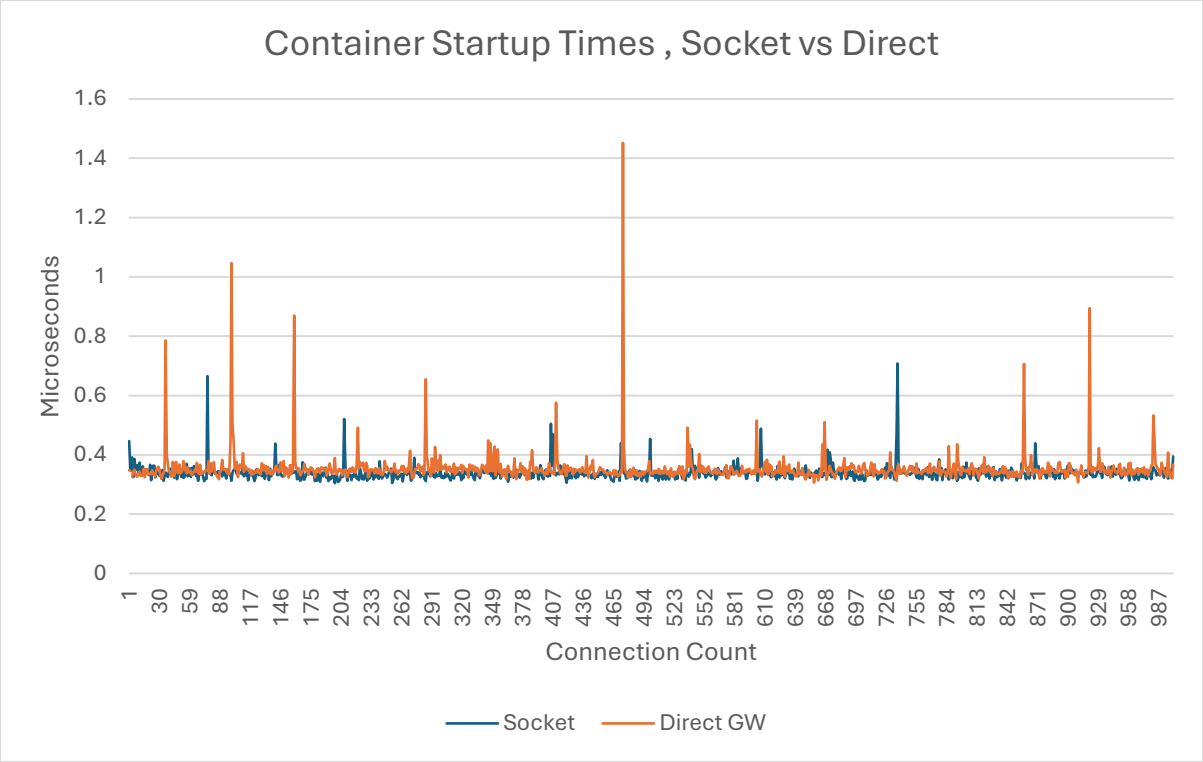

Figure 6

Another significant observation of the results is that the variance in container lifecycle time is significantly higher for the direct connection test run. Figure 6 shows a graph of the socket vs direct container startup times, which illustrates that the container startup times for the direct connections are more volatile than those which use the socket connections. An analysis of the data bears this out - The variance for the direct connections is $3.21ms^2$ compared the socket connection variance of $0.6ms^2$. Using Levene's test to compare the variances shows that the p-value is 0.0055, indicating that the 2 variance values are significantly different. This is further illustrated by measuring the 99^{th} percentile for direct connections, which is 514ms vs the socket connection value of 439ms

In order to further confirm the direct relationship between the container lifecycle time and the time to establish the TCP connection. I ran a number of tests where I artificially introduced a delay to the TCP SYN/SYN-ACK/ACK handshake negotiation, specifically by adding a delay to the SYN-ACK response sent by the TCP server back to the client.

Table 2 shows that the average difference between each container lifecycle increments directly in proportion to the network connection time

	Socket	Direct GW	Direct 10ms Delay	Direct 50ms Delay	Direct 100ms Delay
Average ms	338	353	365	410	461
Diff ms	0	14.9	26.1	71.3	122.1

Table 2

7 Conclusion and Future Work

In this research, I proposed an approach to improving the startup time of a Serverless function container by injecting an existing connection from a pool of connections so that SLAs of OLTP systems could be faster and more predictable. During this research, I proved that this approach was 3.5 times faster at establishing connections than the function directly establishing a connection. Further, I demonstrated that there was a direct correlation between the speed of connection establishment and the overall container lifecycle time, showing that the container executes the serverless function faster by decreasing the time spend during connection establishment. Finally, I demonstrated that the function execution time was more stable and predictable, making SLAs easier to predict and commit to.

However, there were a number of issues encountered during this research which limits its practical application in the real world. Firstly, the fact that TLS connections cannot be injected into the container due to the stack holding state in the user space of the client means that this approach can only be applied to TCP connections which do not have this issue. Secondly, there were issues with how Kubernetes uses network namespaces which also prevents containers which are managed by Kubernetes from using this approach. Finally, there were functional restrictions specifically implemented by OpenFaaS to prevent this pattern from being implemented as it creates a strong dependency between the container and the underlying host.

Future work that could continue this line of research would be to investigate ways of removing the dependency on user-space state during TLS connection negotiation in order to enable passing of a TLS connection from host to container. Other research could be made to find another means of overcoming the network namespace restrictions implemented by Kubernetes so that connections can be passed across network namespace boundaries. Returning the connection to the pool has also not been considered in this research - Is it better for the serverless function to just discard the connection once it has completed its execution and for the connection pool manager to continuously replenish its stock of connections? Or should the serverless function return the connection to the pool via the ICP socket for future re-use?

8 Supervisor Questions

8.1 If I assume your proposed work they what will be the strategy to decide what all services should be in connection pool because we cannot keep all the connection in connection pool which shall increase the cost and resources?

The strategy for determining which services should utilize the connection pooling should be based on a number of criteria:

The first consideration should be whether a service requires a connection in the first place. If it is not a stateful service, there would be no benefit to providing this type of pool management.

The second consideration should be the SLA for the function - Not all serverless functions require a tight, sub-second SLA, and those that do not have this requirement do not need to use this mechanism.

The third consideration should be the resource availability on the FaaS host server and on the target TCP server. Because the Connection Pool manager holds open connections to the TCP server, it is reserving resources on both the local server and the remote TCP server. If the network connections are a scarce resource and are required by other services, then even though there may be a performance benefit, using a connection pool may not be the correct strategy for managing the overall system resources.

8.2 What should be the size of connection pool and how it will define which connect will stay live and which connection should be left behind?

The size of the connection pool should be determined by the number of simultaneous functions each FaaS server can run, which will in turn be determined by the resources required by the serverless function combined with the hardware resources on the server. A server with 64 cores and 2 TB of memory can host many more serverless functions than one which has only 8 cores and 32GB of memory, so it is important to size the connection pool based on the capacity of the server which hosts the serverless functions.

One good indicator for this value would be to use the auto-scaling parameters provided by the FaaS platform. For example, OpenFaaS provides auto-scaling configuration parameters which could be used to indicate parameters (*Autoscaling - OpenFaaS*, 2024) :

com.openfaas.scale.min	The lower boundary for the number of Pods whilst under load.	1
com.openfaas.scale.max	The upper boundary for the number of Pods under maximum load. Or set to same as lower boundary to disable scaling.	20

The connection pool sizes should approximately match the max scale boundary. The only thing to watch here is that these OpenFaaS parameters are global values, so if the FaaS platform has servers of different sizes, then the connection pool sizings may need to be tuned to the specific capacity of each server.

I did not specifically address the management of the lifecycle of the open connections in my research. I did call it out as a potential future research area, with the 2 main options available being to a) return the open connection to the connection pool manager, via the IPC Socket connection or b) discarding the connection, and the Connection Manager would just establish a new connection to the TCP server in the background.

Either of these approaches would be technically feasible but I would favour option b) discarding the connection and establishing a new one for a number of non-performance reasons. The main one would be that each connection would be fresh for each function to use. This is simpler to implement and would maintain the isolation of the network connection to a single function. This would also make it easier to track and trace from an operational point-of-view. Re-using the connection could lead to potential race conditions where the function has handed back the connection but maintains a link to it and re-uses it or closes it unexpectedly.

I would also add a configurable time-to-live (TTL) on the connections in the connection pool, just to ensure that connections are relatively recently established, and have not become stale due to an intermediate device timing out or the server closing the connection without a proper shutdown handshake occurring.

8.3 Discuss the drawbacks of your system if implemented on large scale

There are a number of things to consider if this system was utilised on a large-scale deployment:

The first would be that it would add to the deployment, configuration and operational complexity of the system. An additional component is now involved in the execution of the function, and resourcing, security and configuration would have to be considered for each FaaS server, adding to the DevOps complexity and overhead of the system. Mitigation for this would be to keep the hardware profiles of each FaaS server the same and favour fewer larger servers rather than many smaller servers, thus minimizing the deployment complexity.

The second thing to consider would be that it could be a single point of failure in the system if not implemented correctly. If the only way to retrieve a connection was through the Connection Manager, then this would mean that if the Connection Manager was not available then functions could not execute. The mitigation for this would be to have a backup mechanism where the function could directly establish a connection to the TCP server if it could not retrieve a connection from the Connection Manager

8.4 What shall be the overhead on running this manager

The overhead of running the Connection Manager in the system should be minimal from a CPU and memory point of view. As established in my research, opening a network connection and passing an open file descriptor across an IPC channel generally takes in the order of microseconds, so this should keep processing to an absolute minimum. Beyond establishing network connections to the server, and processing IPC connection requests from local containers, the Connection Manager performs no other processing function.

The most significant system resource usage of the Connection Manager would be the set of open file descriptors which would be used as part of the connection pool. However, these are relatively cheap resources to allocate, and modern Linux systems can handle thousands of open fds without having much of an impact on the system. There is a danger that if the system runs out of open FDs, this could impact other processes on the host, but appropriate configuration of the FD limits should mitigate this.

9 References

Barcelona-Pons, D. *et al.* (2022) 'Stateful Serverless Computing with Crucial', in *ACM Transactions on Software Engineering and Methodology*. Association for Computing Machinery. Available at: https://doi.org/10.1145/3490386.

Bermbach, D., Karakaya, A.S. and Buchholz, S. (2020) 'Using application knowledge to reduce cold starts in FaaS services', in *Proceedings of the ACM Symposium on Applied Computing*. Association for Computing Machinery, pp. 134–143. Available at: https://doi.org/10.1145/3341105.3373909.

CRIU (2025). Available at: https://criu.org/Main_Page (Accessed: 10 April 2025).

Datadog (2023) *The State of Serverless | Datadog*. Available at: https://www.datadoghq.com/state-of-serverless/ (Accessed: 5 August 2025).

Du, D. *et al.* (2020) 'Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting', in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. Association for Computing Machinery, pp. 467–481. Available at: https://doi.org/10.1145/3373376.3378512.

Eismann, S. *et al.* (2021) 'Serverless Applications: Why, When, and How?', *IEEE Software*, 38(1), pp. 32–39. Available at: https://doi.org/10.1109/MS.2020.3023302.

Fireman, D. *et al.* (2024) 'Prebaking runtime environments to improve the FaaS cold start latency', *Future Generation Computer Systems*, 155, pp. 287–299. Available at: https://doi.org/10.1016/j.future.2024.01.019.

Golec, M. *et al.* (2023) 'Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions'. Available at: https://doi.org/10.1145/3700875.

Golec, M. *et al.* (2024) 'ATOM: AI-Powered Sustainable Resource Management for Serverless Edge Computing Environments', *IEEE Transactions on Sustainable Computing*, 9(6), pp. 817–829. Available at: https://doi.org/10.1109/TSUSC.2023.3348157.

Govindaraj, V. (2024) 'Cloud Migration Strategies for Mainframe Modernization: A Comparative Study of AWS, Azure, and GCP', *International Journal of Computer Trends and Technology*, 72(10), pp. 57–65. Available at: https://doi.org/10.14445/22312803/ijctt-v72i10p110.

de Heus, M. *et al.* (2022) 'Transactions across serverless functions leveraging stateful dataflows', *Information Systems*, 108, p. 102015. Available at: https://doi.org/10.1016/J.IS.2022.102015.

Jonas, E. *et al.* (2019) *Cloud Programming Simplified: A Berkeley View on Serverless Computing*.

Joosen, A. *et al.* (2023) 'How Does It Function? Characterizing Long-term Trends in Production Serverless Workloads', in *SoCC 2023 - Proceedings of the 2023 ACM Symposium on Cloud Computing*. Association for Computing Machinery, Inc, pp. 443–458. Available at: https://doi.org/10.1145/3620678.3624783.

Kohli, S. *et al.* (2024) 'Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts', in *EuroSys 2024 - Proceedings of the 2024 European Conference on Computer Systems*. Association for Computing Machinery, Inc, pp. 298–316. Available at: https://doi.org/10.1145/3627703.3629556.

Lannurien Vincent and D'Orazio, L. and B.O. and B.J. (2023) 'Serverless Cloud Computing: State of the Art and Challenges', in A. and G.S.S. and B.R. Krishnamurthi Rajalakshmi and Kumar (ed.) *Serverless Computing: Principles and Paradigms*. Cham: Springer International Publishing, pp. 275–316. Available at: https://doi.org/10.1007/978-3-031-26633-1_11.

Live migration - CRIU (2023). Available at: https://criu.org/index.php?title=Live_migration&mobileaction=toggle_view_desktop (Accessed: 10 April 2025).

Mampage, A., Karunasekera, S. and Buyya, R. (2022) 'A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions', *ACM Computing Surveys*, 54(11s). Available at: https://doi.org/10.1145/3510412.

McDonald, P. (2008) *Google App Engine Blog: Introducing Google App Engine + our new blog*. Available at: https://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html (Accessed: 5 August 2025).

Mohan, A. *et al.* (no date) *Agile Cold Starts for Scalable Serverless*.

network_namespaces(7) - Linux manual page (2024). Available at: https://man7.org/linux/man-pages/man7/network_namespaces.7.html (Accessed: 24 July 2025).

Nguyen, H.D. *et al.* (2019) 'Real-time Serverless: Enabling application performance guarantees', in *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*. Association for Computing Machinery, Inc, pp. 1–6. Available at: https://doi.org/10.1145/3366623.3368133.

OpenFaaS CE - OpenFaaS (2024). Available at: https://docs.openfaas.com/deployment/kubernetes/ (Accessed: 24 July 2025).

openfaas/faasd: Lightweight and portable version of OpenFaaS (2025). Available at: https://github.com/openfaas/faasd (Accessed: 30 July 2025).

Proposal: Adding mount options to functions. · Issue #320 · openfaas/faas (2022). Available at: https://github.com/openfaas/faas/issues/320 (Accessed: 15 July 2025).

Rescorla, E. (2018) *RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3*. Available at: https://datatracker.ietf.org/doc/html/rfc8446 (Accessed: 30 July 2025).

Schirmer, T. *et al.* (2024) 'FUSIONIZE++: Improving Serverless Application Performance Using Dynamic Task Inlining and Infrastructure Optimization', *IEEE Transactions on Cloud Computing* [Preprint]. Available at: https://doi.org/10.1109/TCC.2024.3451108.

Services, Load Balancing, and Networking | Kubernetes (2024). Available at: https://kubernetes.io/docs/concepts/services-networking/ (Accessed: 24 July 2025).

Silva, P., Fireman, D. and Pereira, T.E. (2020) 'Prebaking functions to warm the serverless cold start', in *Middleware 2020 - Proceedings of the 2020 21st International Middleware Conference*. Association for Computing Machinery, Inc, pp. 1–13. Available at: https://doi.org/10.1145/3423211.3425682.

tc(8) - Linux manual page (2025). Available at: https://man7.org/linux/man-pages/man8/tc.8.html (Accessed: 7 August 2025).

Xu, Z. *et al.* (2019) 'Adaptive function launching acceleration in serverless computing platforms', in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. IEEE Computer Society, pp. 9–16. Available at: https://doi.org/10.1109/ICPADS47876.2019.00011.