The Cold Start Problem: How container life cycle influences serverless functions

Ben Civjan Diego Chahuan

May 29, 2022

Contents

1 Introduction

Serverless computing has gained a huge amount of traction among developers and DevOps professionals in recent years because of its simplicity and cost efficiency in addition to all the other benefits of cloud computing. It coincides with and complements the paradigm shift from the monorepo to microservice, where developers are increasingly breaking up their applications into smaller parts for better scalability, reliability, and easier code maintainability.[Gar]

The first company to introduce modern serverless computing was Amazon Web Services with the invention of Lambda Functions in 2014. Since then both Microsoft and Google have entered the serverless market, with Azure Functions in 2016 and Google Cloud Functions in 2018, respectively.[Fru]

The figure below shows a steady increase in Google searches regarding serverless computing. The beginning of the incline, beginning at around 2015, can be attributed to the release of Lambda Functions. With more competitors entering the market, serverless appears to have a very promising future in the developer community.

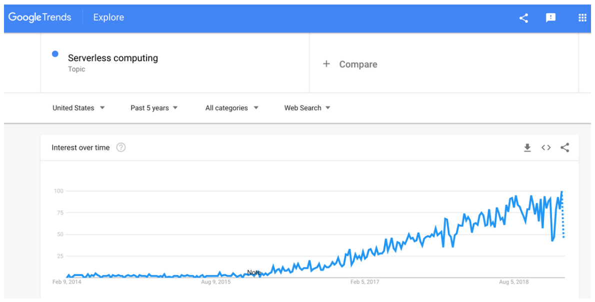

Figure 1: Google searches of 'Serverless computing' over time [Unk]

Despite the benefits that serverless computing offers, one of the biggest drawbacks of using this architecture is performance. This comes from something known as a *cold start*. This article will describe what this issue is, dive into what exactly causes it, why and how container lifecycle has an impact, and provide some solutions for solving the problem.

After reading this article, a developer with little to no experience with deploying a serverless architecture will understand the pros and cons of doing so, will understand how serverless works behind the scenes, and will be able to understand and address an important drawback of the popular design.

2 Benefits of a Serverless Architecture

Before solutions to the cold start problem can be addressed and analyzed, one must understand the benefits of serverless computing and why it has become such a popular architecture used in application development. This section will provide the necessary context for why improvements to serverless are worth researching: to make an already useful model of computing even better.

2.1 A Faster and Cheaper Way to Scale

The serverless or FaaS architecture is one of the latest models of cloud computing. Once companies began realizing that building and maintaining their own servers was becoming difficult to manage because it was tough to scale and hard to maintain good availability, the need for cloud computing was born. The movement started with virtual machines hosted on other companies servers, which allowed for a customer to have a fully managed, configured, and easily scalable server without any of the maintenance that comes with owning and operating your own servers. Later with the popularization of microservices and the creation of virtual containers, keeping a machine running became unnecessary and expensive. The idea of minimizing the amount of compute time that your application must use evolved into the concept of serverless computing: Separating an application into several independent and stateless functions that only occupy memory and CPU space when run, saving computational resources.

Figure 2 is an example provided by AWS to show how lambda functions can be used in a simple ”to-do list” application. The diagram shows how the application can be decomposed into each specific operation: addTodo, completeTodo, deleteTodo, etc.

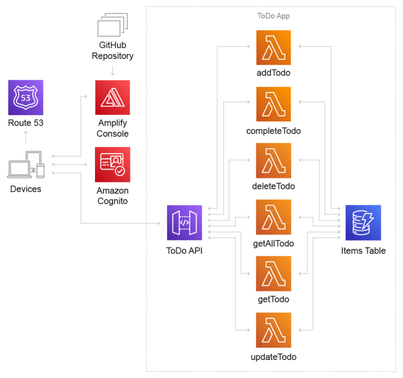

Figure 2: AWS example of serverless with lambda functions[Ser]

2.2 Functionality and Advantages

“After triggering a function by an event, [the] function will be deployed on a container and given all the resources needed to execute it. When the function executed completely and there were no subsequent requests, all the allocated resources will be released.” [VFA20] This triggering can result from an HTTP request, IoT device sending information, or any other networking event. The repeated starting and stopping of containers is what makes serverless more affordable and efficient.

This allows servers to scale to zero if 0 requests are being processed. This makes building software cheaper since the customer is only charged for what resources are actually being used. A second advantage is that this hides the complexity that maintaining your own server concurs and allows a business to focus more resources on their product.

3 The Cold Start Problem and the Influence of Containers

The cold start problem occurs “when a request comes in and no idle container can be found for the execution of the target function, then a new container needs to provisioned. In that case, the request incurs an extra latency - the cold start latency”[BKB20].

The amount of time taken to execute a function with a serverless architecture is heavily dependent on the containerization software used. The steps can be seen in Figure 3, where a container needs to be scheduled to start, pulled from the registry, and initialized all before being able to execute code. Pulling an image can take hundreds of milliseconds and initializing a container can take a similar amount of time. This creates a delay that is non-existent in traditional servers.

The usage of containers also limits the amount of concurrency that the server can achieve, since “maximum throughput of creating empty Docker containers is about 200-230 containers per second with the machine” [QIN+20]. Concurrency vulnerability issues may come into play in big scale applications or IoT devices where events that trigger lambda functions are constantly happening.

Resuming a stopped container is 100x or more faster than creating a new container, so it is generally preferred over a complete reinitialization.[QIN+20] However, stopping doesn’t release memory so only a relatively small amount of containers can exist at once in a machine’s memory. This may impact performance when RAM memory is near capacity because it will need to do frequents swaps with the hard drive, an operation that is extremely inefficient. It also makes it impossible to scale to 0 because resources are never cleared.

3.1 Concurrency

The usage of containers as the driving technology for serverless also creates potential for runtime issues when dealing with concurrency. In the paper *Evaluation of Production Serverless Computing*

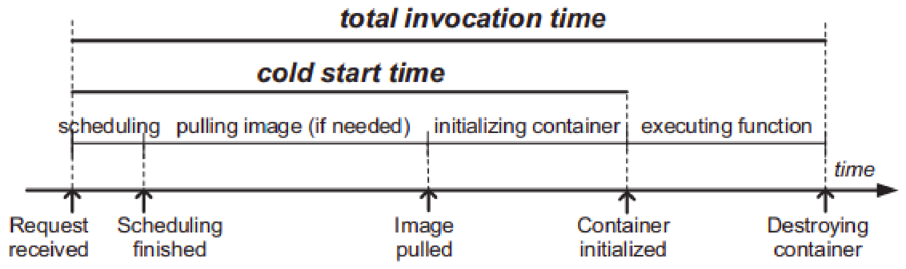

Figure 3: Container provisioning timeline in function initialization process [QIN+20]

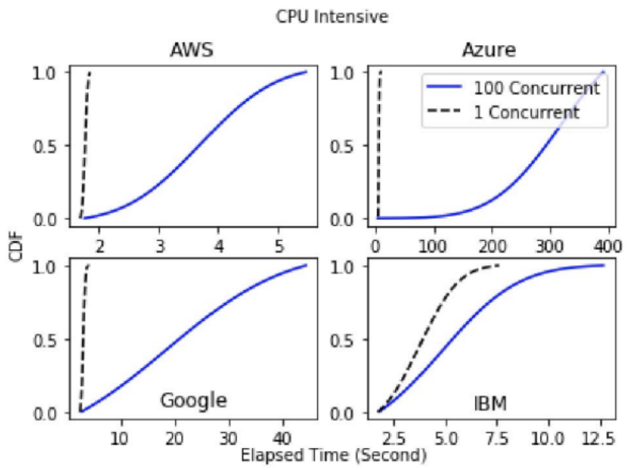

Figure 4: Runtime of 4 major serverless providers with concurrency stress-test [LSF18]

Environments, an experiment that stress-tested concurrency among 4 major serverless providers was conducted. "The results with 1 concurrent invocation which is non-parallel invocation are consistent whereas the results with 100 invocations show the overhead of between 28the total execution time."[LSF18] The results of the experiment are shown in Figure 4. The sharp increase in runtime when faced with concurrency suggests that once all available warm containers are in use, additional executions of the function are extremely expensive. The architecture must decide whether to share resources among already running containers or create new ones, with both options resulting in a slower program.

3.2 A Case Study of AWS

In a 2020 IEEE article titled *A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing*, a deeper analysis of initialization latency in AWS Lambda was conducted. The experiment was conducted by making 1000 lambda requests with 20 workers. The findings were that "fresh-start and restart incur 1000 times and 10000 times more delay than unpause state respectively."[SA20] The results are displayed in Figure 5, where there is a clear latency difference between the types of initializations.

With proper understanding of container software and lifecycles it is possible to optimize performance and throughput of a serverless machine. Some solutions that exploit the traits of containers for optimization purposes are described in the next section.

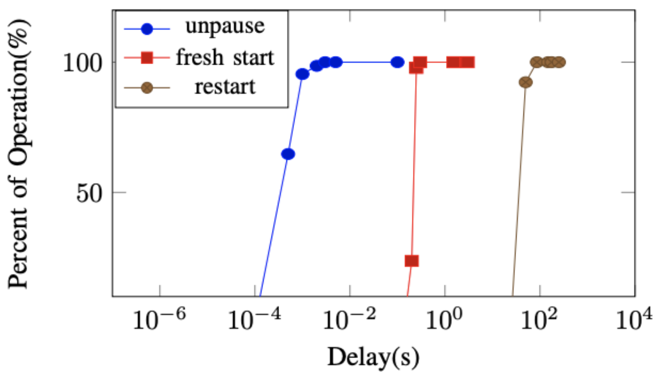

Figure 5: Runtime Latency of unpause, fresh start, and restart operations in AWS OpenLambda [SA20]

4 Solutions

There will be 5 different approaches discussed to counteract the cold start problem.

4.1 Näive Approach

The first approach is called pinging. The strategy involves periodically 'pinging' the function with a request to keep it active. This uses the fact that generally FaaS providers keep some containers warm a fixed amount of time. It avoids cold functions by making them permanently active.

The pros of this approach are that it simple to implement, it requires few lines of code and keeps the serverless function warm. The disadvantages are that it is resource intensive because the container is never booted out of memory and the processes can always be running. This defeats one of the core objectives of serverless, which is to use as few resources as possible. Also the container idle-time may not be constant, so you would need to check if your pinging job is running at the correct frequency.

4.2 Machine learning to determine idle time

This approach uses a reinforcement learning neural network to calculate the optimal time that the function would need to stay idle for best performance. This is accomplished by creating a function that rewards a few cold starts but penalizes high memory usage. At the end this can give you up to 12.73% less memory [VFA22] without creating more cold starts.

The advantage of using ML is that you get the same performance as keeping the container idle for about 10 minutes but use much less memory in the process. The disadvantage is that aggregating an neural network that can scale to the size of some of the providers may be hard. As well as the actual implementation.

4.3 Improving container start up time

One way to improve container start up time is with application sandboxing. This approach plays with the application isolation. Here there must exist a strong level of isolation between applications but functions for each application do not require isolation [VFA20].

This allows for starting processes in the application container instead of a full new container and allows for the sharing of dependencies. The trade off of this is speed vs memory efficiency.

4.4 Pooling

A strategy known as pooling can be used to keep a certain number of containers warm in order to remedy the initialization time. Pooling is described as "maintain[ing] a pool of warm pods that will stand-by and be immediately available to functions with increasing demand, eliminating the cold start overhead by provisioning the pods beforehand." [LG19] The key difference between this and the 'Näive Approach' is that the pool of containers are orchestrated with Knative, "an open source initiative aiming to provide a platform for developing container applications on top of the Kubernetes container orchestration platform." [LG19] This leads to more efficient usage of server resources and makes it a much more realistic solution than pinging. However, a drawback of this is that it requires the developer to have much more control over the serverless architecture in order to specify pooling size and structure.

4.5 Prebaking

The prebaking approach attempts to mitigate the cold start problem by taking 'snapshots' of each lambda function whenever a new version is pushed by the developer. A simple snapshot strategy would be to take a snapshot of the function immediately after the start-up procedure. A more fine-tuned approach is possible, but requires "requires in-depth knowledge about the runtime, such as when the start-up procedure reaches the right balance between progress and the amount of state generated." [SFP20] New invocations of the function can then begin execution at the point when the snapshot was taken, effectively skipping the common start-up costs.

5 Conclusion and Reflections

In conclusion, the cold start problem is deeply linked with the lifecycles of the docker containers and how the different stages provide different trade-offs. Where the FaaS provider needs to find a solution that gives good performance without overusing the server resources and keeping the advantages that serverless provides. Currently, several solutions have been attempted but none are perfect or give a permanent solution to the problem. In the near future FaaS will likely continue to gain popularity, but the cold start delay will be a deterrent for developers that want to create applications using this technology.

It may be possible in the future to create a new container engine with the same interface as Docker that can create containers at little cost. A new engine tailored to serverless should include fast initialization, low pausing cost and a cache of commonly used libraries speeding the whole process up. In the meantime, current software is being optimized in creative ways to minimize the expensive parts of the container lifecycle while also minimizing the computing resources used.

References

[LSF18] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. "Evaluation of Production Serverless Computing Environments". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 442–450. DOI: 10.1109/CLOUD.2018.00062.

[LG19] Ping-Min Lin and Alex Glikson. "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach". In: *CoRR* abs/1903.12221 (2019). arXiv: 1903.12221. URL: http://arxiv.org/abs/1903.12221.

[BKB20] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. "Using application knowledge to reduce cold starts in FaaS services". In: *SAC '20: Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic, Mar. 30–Apr. 3, 2020). 2020.

[QIN+20] Shijun QIN et al. "Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing". In: *2020 IEEE International Conference on Joint Cloud Computing (JCC)* (Oxford, UK, Aug. 3–6, 2020). 2020.

[SFP20] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. "Prebaking Functions to Warm the Serverless Cold Start". In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450381536. DOI: 10.1145/3423211.3425682. URL: https://doi.org/10.1145/3423211.3425682.

[SA20] Khondokar Solaiman and Muhammad Abdullah Adnan. "WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing". In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 2020, pp. 144–153. DOI: 10.1109/IC2E48712.2020.00022.

[VFA20] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. "Cold Start in Serverless Computing: Current Trends and Mitigation Strategies". In: *2020 International Conference on Omni-layer Intelligent Systems (COINS)* (Barcelona, Spain, Aug. 31–Sept. 1, 2020). 2020.

[VFA22] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. "Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach". In: *IEEE Internet of Things Journal* 15 (2022), pp. 1–11.

[Fru] Josh Fruhlinger. *What is serverless? Serverless computing explained.* URL: https://www.infoworld.com/article/3406501/what-is-serverless-serverless-computing-explained.html.

[Gar] Ekta Garg. *Moving from monolithic to microservices? What, why and how to design microservices.* URL: https://medium.com/@_ektagarg/moving-from-monolithic-to-microservices-what-why-and-how-to-design-microservices-858aed666357.

[Ser] Amazon Web Services. *Serverless on AWS: Build and run applications without thinking about servers.* URL: https://aws.amazon.com/serverless/.

[Unk] Unknown. *Adoption and Market Data.* URL: https://www.ematop3.com/serverless-research-highlights.html.