

## Evaluación de la inicialización y el arranque en frío de los lenguajes de programación en una plataforma serverless. Amazon Web Services como caso de estudio

Matías Rodríguez, Nelson Rodríguez, María Murazzo,

Departamento de Informática, Facultad de Ciencias Exactas Físicas y Naturales, Universidad  
Nacional de San Juan, San Juan, Argentina

matiasdaroni@gmail.com, nelson@iinfo.unsj.edu.ar, maritemurazzo@gmail.com

**Resumen.** Serverless Computing es una arquitectura en el Cloud, alternativa al modelo tradicional. Ofrece numerosas ventajas sobre el modelo monolítico, como aportar agilidad, innovación, escalado automático, flexibilidad en el desarrollo y una mejor evaluación y control de los costos. Surgió como una evolución de microservicios corriendo en contenedores e implementando funciones, por lo cual a veces se lo denomina función como servicio.

En el presente trabajo se evalúa el comportamiento de distintos lenguajes de programación en la inicialización y el arranque en frío, bajo un enfoque serverless en la plataforma Amazon Web Services. Debido a que la comparación de los lenguajes se puede llevar a cabo en diferentes aspectos o tomando diferentes tipos de métricas, se consideró oportuno realizar las pruebas mediante operaciones CRUD. Esto además permite analizar el comportamiento de los mismos al inicializar la base de datos. Surgen por lo tanto una serie de interrogantes como: ¿La inicialización de los lenguajes de programación son todos iguales?, ¿La inicialización de la base de datos con los lenguajes de programación es independiente del lenguaje usado?, ¿El cold start es equivalente en los distintos lenguajes? ¿El impacto del mismo es relevante cuando la cantidad de requerimientos es elevada? ¿Cuál es la combinación de lenguajes de programación más conveniente para operaciones CRUD?.

Mediante las pruebas realizadas se pudieron evaluar el comportamiento de los lenguajes de programación y contestar algunos de los interrogantes arriba mencionados.

**Keywords:** Serverless Computing, FaaS, Programming Language, Cloud Computing

### 1. Introducción

El término Cloud Computing entró en uso popular en 2008, aunque la práctica de proporcionar acceso remoto a las funciones informáticas a través de redes se remonta a los sistemas de tiempo compartido de mainframe de los años 1960 y 1970. La Virtualización creada por IBM, permitió que múltiples sistemas virtuales se ejecuten sobre un solo sistema físico [1]. Luego surge Internet de forma global y su expansión comenzó cuando en 2002, Amazon introdujo sus servicios minoristas basados en la web. Fue la primera gran empresa en pensar en usar solo el 10% de su capacidad (que era común en ese momento) como un problema a resolver [2] [3].

Siguiendo la evolución observada en la historia de la contenerización, los servicios en la nube se han adaptado para ofrecer contenedores de mejor ajuste que requieren

menos tiempo para cargar (arranque) y proporcionar mayor automatización en el manejo (orquestración) de contenedores en nombre del cliente [4]. La Computación Serverless promete lograr la automatización completa en la gestión de contenedores.

El modelo de computación serverless es impulsado por eventos en el que los recursos informáticos se proporcionan como servicios escalables. En el modelo tradicional se cobra un costo fijo y recurrente por los recursos informáticos del servidor, independientemente de la cantidad de trabajo realizado por el servidor. Sin embargo, la implementación Serverless ha superado esta deficiencia, ya que se paga solo por el uso del servicio y no se cobra por el tiempo de inactividad.

En este paradigma emergente, las aplicaciones de software se descomponen en múltiples funciones independientes sin estado [5] [6]. Las funciones solo se ejecutan en respuesta a acciones desencadenantes (como interacciones de usuario, eventos de mensajería o cambios en la base de datos), y se pueden escalar de forma independiente y pueden ser efímeras (pueden durar una invocación) y están completamente administrados por el proveedor de Cloud.

Los principales proveedores de nube han propuesto diferentes plataformas informáticas sin servidor como AWS Lambda, Microsoft Azure Functions, Google Functions, IBM Cloud Functions, Cloudflare Worker, Alibaba Function Compute. Dichas plataformas facilitan y permiten que los desarrolladores se centren más en la lógica de negocios, sin la sobrecarga de escalar y aprovisionar la infraestructura [7].

Castro et al [8], ofrecen una definición basada en las características: “La informática serverless se puede definir por su nombre, que es pensar (o preocuparse) menos por los servidores. Los desarrolladores no necesitan preocuparse por los detalles de bajo nivel de administración y escalado de servidores, y solo pagan cuando procesan solicitudes o eventos”. Luego la define como: “La informática serverless es una plataforma que oculta el uso del servidor a los desarrolladores y ejecuta código que escala bajo demanda automáticamente y facturado solo por el tiempo que se ejecuta el código”.

En la mayoría de los casos, se pueden escribir funciones en el lenguaje que el programador considere más adecuado (Node.js, Python, Go, Java y más) y utilizar herramientas de contenedor y serverless, como AWS SAM o la CLI de Docker, para compilar, probar e implementar las funciones [9].

Por otro lado, especialistas de Expert Market Research pronostican que el mercado global de computación serverless crecerá en el período de pronóstico de 2022-2027 a una tasa compuesta anual del 22.2% [10].

Un modelo basado en funciones es adecuado para ráfagas, uso de CPU intensivo, cargas de trabajo granulares. Actualmente, los casos de uso de FaaS varían ampliamente, incluido el procesamiento de datos, el procesamiento de flujo, la computación de borde (IoT) y la computación científica [3].

Serverless cubre una amplia gama de tecnologías, que se pueden agrupar en dos categorías: Backend-as-a-Service (BaaS) y Functions-as-a-Service (FaaS).

Backend-as-a-Service permite reemplazar los componentes del lado del servidor con servicios listos para usar. Algunos ejemplos son los sistemas de autenticación remota, la administración de bases de datos, el almacenamiento en el cloud.

Existen diversos desafíos, oportunidades y problemas a resolver, entre ellos la experiencia del desarrollador [8], Interoperabilidad, testing, composición de funciones,

seguridad, administración del ciclo de vida, administración de requerimientos no funcionales, performance, optimización del overhead, ingeniería para costo-performance, entre otros [7].

Idealmente, sería deseable tener una sobrecarga mínima al invocar las funciones. Sin embargo, cuando la plataforma necesita activar la primera instancia, los recursos subyacentes aún necesitan tiempo para la inicialización. Este arranque también ocurre cuando el escalador automático aprovisiona instancias adicionales para controlar el tráfico. El tiempo de inicialización es inevitable para cada instancia de función e introduce un retraso hasta que la instancia pueda responder a las solicitudes. Este problema es muy común en plataformas serverless y se conoce como el problema de arranque en frío. Aunque la oferta de FaaS generalmente sufre de arranques en frío, la sobrecarga en la que incurre cada plataforma varía según la implementación subyacente de las funciones [11]

## **2. Trabajos relacionados**

En el desarrollo de software basado en Cloud, los recursos alquilados siempre se pueden mantener activados para servir a la ejecución de la aplicación, que no tiene problemas de arranque en frío para las ejecuciones de aplicaciones. Además, los desarrolladores pueden seleccionar de forma flexible la capacidad de los recursos y configurar el tiempo de servicio de los recursos.

A diferencia, en desarrollo de software basado en serverless, el proveedor es responsable de la administración del entorno de tiempo de ejecución. La ventaja de este tipo de gestión de recursos unificados es responder a cualquier carga de trabajo en ráfaga. Estos entornos de tiempo de ejecución se activan cuando se activan las aplicaciones. Cuando los entornos requeridos no están activos, las aplicaciones pueden enfrentar el problema del arranque en frío, lo que introduce un largo tiempo de preparación. Ha habido muchos esfuerzos para aliviar este problema [12,13, 14, 15, 16, 17]. Sin embargo, ninguno de estos trabajos analiza el comportamiento de los lenguajes de programación en esta problemática.

Un trabajo publicado por [18] es muy claro en la explicación del problema, afirmando lo siguiente: Serverless Computing es el último modelo de computación en la nube. En este modelo la plataforma serverless ofrece elasticidad instantánea por solicitud. Tal elasticidad generalmente se produce a costa del problema de los "arranques en frío". Este fenómeno está asociado con un retraso que se produce debido a la provisión de un contenedor en tiempo de ejecución para ejecutar las funciones. Poco después de que Amazon introdujera este modelo informático con la plataforma AWS Lambda en 2014, Varias plataformas comerciales y de código abierto comenzaron a adoptar y ofrecer esta tecnología. Cada plataforma tiene su propia solución para hacer frente a los arranques en frío. La evaluación del rendimiento de cada plataforma bajo la carga y los factores que influyen en el problema del arranque en frío ha recibido mucha atención en los últimos años.

## **3. Metodología**

La problemática abordada es la evaluación del comportamiento de los lenguajes de programación para la inicialización y el arranque en frío, en una plataforma serverless, tomando como caso de estudio a Amazon Web Services.

Para llevar a cabo los estudios de laboratorio pertinentes sobre diversas variables en la ejecución y puesta en marcha de Funciones Serverless (FaaS), se realizarán las siguientes actividades:

- Análisis y estudio de la plataforma o servicio AWS Lambda.
- Estudio y funcionamiento detallado de las bases de datos disponibles en AWS.
- Determinación de los lenguajes a comparar haciendo uso de funciones.
- Determinación de los escenarios a utilizar que permitan contrastar los valores resultantes.
- Estudio comparativo de los diferentes parámetros y valores obtenidos.

Para el desarrollo de la investigación se procedió como se muestra en la figura 1, la cual consiste en un esquema que sintetiza la labor realizada.

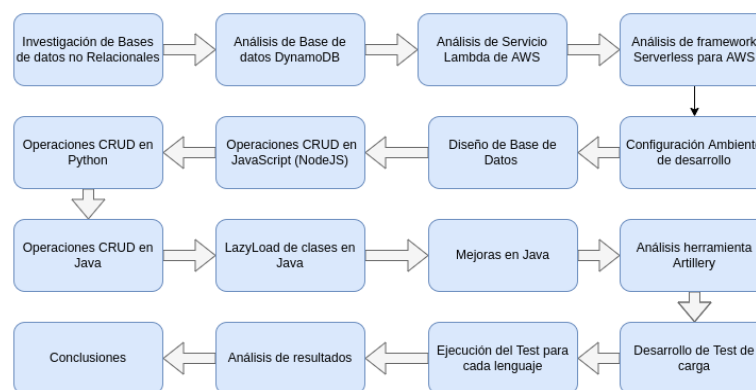


Figura 1 – Esquema del procedimiento del proceso de investigación

Para la investigación se tomó la decisión de desarrollar una API de formularios, utilizando los servicios de AWS que permiten un desarrollo serverless. Se consideró a una API que permita el manejo de distintos formularios dinámicos y que, a su vez, conserve la información pertinente a estos junto con sus respuestas. Lo anterior se pensó adecuado para reproducir un problema del mundo real en el que se necesite de recursos de software escalables cuyos objetivos sean los de soportar grandes volúmenes de usuarios sin afectar el rendimiento del sistema.

Se dispuso realizar una API de formularios, para la comparativa de los distintos lenguajes de programación en un entorno serverless.

La función de esta API es la de crear formularios dinámicos y almacenar toda la información requerida por estos, incluyendo las respuestas de los usuarios. Para realizar la evaluación de los lenguajes bajo este contexto, solamente se necesitó de un formulario. Para la evaluación se implementó un extracto del formulario utilizado para el Censo Nacional de Población, Hogares y Viviendas 2022, dado que este tiene muchos campos y datos de variado tipos.

#### 4. Base de datos y ambiente de desarrollo

Con la finalidad de almacenar los datos necesarios para la representación de formularios y sus respuestas, se optó por utilizar el servicio DynamoDB de AWS ya que permite realizar la investigación bajo el enfoque serverless debido a que este servicio es totalmente administrado y evita las cargas administrativas que supone tener que utilizar y escalar bases de datos. También este servicio dispone de la potencia necesaria para crear tablas que permiten almacenar y recuperar miles y hasta millones de datos y solicitudes.

En esta investigación se propuso imitar de la mejor forma al tráfico de un aplicativo real y de uso constante, por lo que este servicio resulta fundamental para poder replicar el contexto deseado en el cual se evaluará a los lenguajes de programación elegidos.

Para el desarrollo de las APIs de formularios en los distintos lenguajes sobre la plataforma de AWS, se hizo uso de varias herramientas que permiten y facilitan este desarrollo. Las mismas son:

- Github
- Serverless Framework
- AWS CLI
- NodeJS y NPM
- Python y PIP
- Maven
- Java SDK

Algunas de las mencionadas anteriormente son dependencias necesarias para el desarrollo de las APIs dependiendo cada lenguaje de programación. Por ejemplo para JavaScript con NodeJS se necesita instalar NodeJS y NPM, para Python se requiere de Python y PIP, por último para desarrollar con Java se necesita Maven y el SDK de Java.

El desarrollo de este trabajo fue realizado en una notebook personal bajo la distribución Ubuntu 20.04 del sistema operativo Linux.

#### 5. Resultados obtenidos

Para realizar el análisis comparativo entre los lenguajes de programación seleccionados, con el caso de estudio elegido, se realizaron varias experiencias a través de variaciones de test de inicialización. Cada uno de estas variantes está orientado a un objetivo específico y fueron desarrollados y ejecutados mediante la herramienta open source Artillery [19]. De esta manera se evaluaron diversos aspectos, tales como: tiempos de inicialización para las primeras solicitudes a las funciones lambdas, tiempos de ejecución en frío, tiempos de ejecución en caliente y número de solicitudes

El test de inicialización, con todas las variantes mencionadas, permitió el análisis de la performance de los lenguajes bajo distintas situaciones.

Para llevar a cabo el test de inicialización se optó por ejecutarlos con 10 GB de memoria. Este es el tamaño máximo de memoria ofrecido por AWS en el servicio Lambda, lo cual posibilita utilizar la mayor potencia de CPU permitida por este.

### 5.1 Test de iniciación

El test de inicialización tiene por objetivo probar las operaciones CRUD de tal forma que siempre se produzca la inicialización del contexto de ejecución de cada función lambda. Cuando esto ocurre AWS nos brinda en sus métricas una variable de tiempo de inicialización que indica cuánto le tomó a la plataforma preparar la infraestructura y dependencias necesarias para permitir la ejecución del código dentro de la lambda que fue escrito por el consumidor del servicio.

Se realizaron cinco ejecuciones en frío para cada lenguaje y operación CRUD, esto se hizo mediante la ejecución del flujo del test mencionado en el capítulo anterior. Es importante remarcar que estas ejecuciones se realizaron cada una hora para asegurar que AWS dio de baja la infraestructura FaaS retornando cada función a un estado frío. Cada vez que se hizo una ejecución en frío, también se realizó una segunda ejecución de forma inmediata para tomar el tiempo de ejecución en caliente, lo que permitió realizar una comparativa entre los tiempos de ejecución en frío y caliente.

### 5.2 Tiempos de inicialización

En la tabla 1, se presentan los promedios obtenidos de los tiempos de inicialización correspondientes a las operaciones CRUD por cada lenguaje de programación.

**Tabla 1.** Promedio de tiempos de inicialización por cada operación CRUD y lenguaje de programación

Lenguaje	Create	Read	Update	Delete
JavaScript (NodeJS)	418,232	431,04	413,586	442,15
Python	384,902	400,43	361,624	411,2
Java	1623,528	1630,52	1526,12	1577,302

Teniendo en cuenta todas las operaciones CRUD, Se puede ver una clara diferencia, donde la operación read de Java es la que mayor tiempo de inicialización registra con 1630.52 ms. En cuanto a NodeJS y Python la operación más deficiente es delete con tiempos de 442.15 ms. y 411.2 ms. respectivamente. Estas diferencias pueden apreciarse en la figura 2 aunque no son significativas. Siguiendo el mismo razonamiento la operación update es la de mejor rendimiento a la hora de inicializar el contexto de ejecución para una lambda, cualquiera sea el lenguaje.

Como se mencionó anteriormente, también se obtuvieron resultados en cuanto a los tiempos de ejecución en frío y de una segunda ejecución en caliente. Las tablas 2 y 3 y 4 muestran las diferencias entre los tiempos de ejecución en frío y en caliente para cada una de las operaciones CRUD y por cada lenguaje. Las celdas sombreadas corresponden a la operación CRUD que mayor mejora logra. Se puede apreciar que en Java es donde las diferencias son mayores. Por ejemplo, con delete, la ejecución pasa de tardar 581.494 ms. en frío a 84.654 ms. en caliente. Es decir, Java es el lenguaje con mayor discrepancia entre esos tiempos de ejecución, obteniendo una mejora en

promedio del 84%. Dicho de otro modo, Java es el que reduce los tiempos de forma más abrupta. También NodeJS y Python mejoran al ejecutarse en caliente, encontrando las mayores ganancias en Read y Delete, respectivamente, de aproximadamente un 48% en promedio.

**Tabla 2.** Tiempo de ejecución en NodeJS, por cada operación CRUD

Estado de tiempo de Ejecución	Create	Read	Update	Delete
Frio	76.698	74.592	84.04	77.768
Caliente	51.238	39.248	45.536	41.066
Diferencia	25.46	35.294	38.504	36.702
Porcentaje de Mejora	33.195112	47.347803	45.816278	47.194218

A continuación, se muestra la tabla 3, con los valores de ejecución en frío y en caliente de Python.

**Tabla 3.** Tiempo de ejecución en Python, por cada operación CRUD

Estado de tiempo de Ejecución	Create	Read	Update	Delete
Frio	60.734	49.752	65.738	43.97
Caliente	32.582	27.388	38.472	22.566
Diferencia	28.152	22.364	27.266	21.404
Porcentaje de Mejora	46.352948	44.950956	41.476771	48.6786445

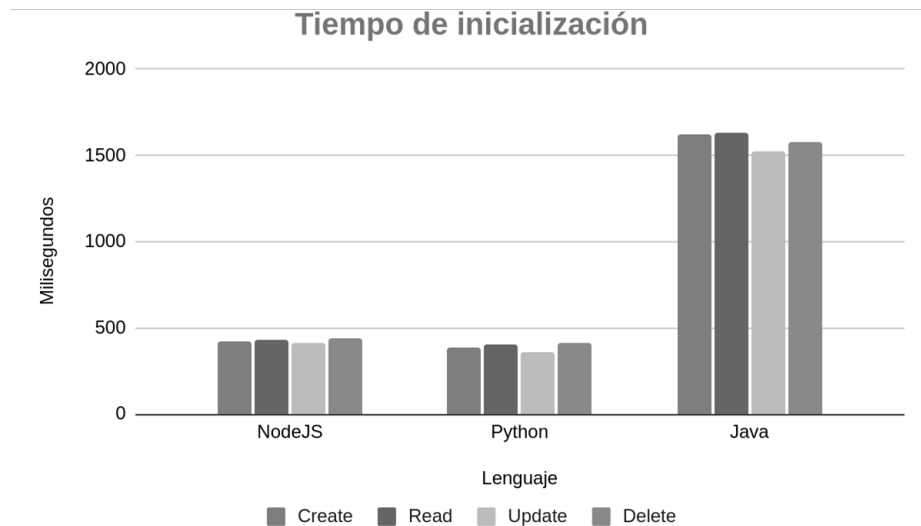
Y por último se muestran los resultados obtenidos luego de ejecutar las pruebas en el lenguaje de programación Java.

**Tabla 4.** Tiempo de ejecución en Java, por cada operación CRUD

Estado de tiempo de Ejecución	Create	Read	Update	Delete
Frio	701.192	664.628	644.328	581.494
Caliente	116.638	98.102	116.89	84.654
Diferencia	584.554	566.526	527.438	496.84
Porcentaje de Mejora	83.365754	85.239562	81.858618	85.441982

Por último se muestra en la figura 2, un resumen de los resultados de todas las ejecuciones realizadas, aunque algunas de las diferencias no son significativas.





**Figura 2.** Comparativa de los tiempos de inicialización de operaciones CRUD de los diferentes lenguajes de programación.

## 6. Conclusiones y Futuros trabajos

En este trabajo se desarrolló una API Rest con el objetivo de realizar operaciones CRUD para la carga de un formulario y sus respuestas. Esta API fue desarrollada en tres lenguajes diferentes: Javascript con NodeJS, Python y Java. Con estas operaciones se investigó, en general, la variación de la performance, la latencia de los tiempos de inicialización, la escalabilidad y las implicaciones del tamaño de memoria configurado en las funciones. Se investigó cómo afecta la selección de los lenguajes de programación a la hora de realizar operaciones CRUD de una API Rest con una serie de test automáticos.

El lenguaje de programación con mejores tiempos a la hora de levantar el contexto de ejecución de una función lambda es Python, obteniendo los mejores tiempos para cualquiera de las operaciones CRUD. De igual forma, las diferencias entre Python y NodeJS son realmente insignificantes. La elección entre cualquiera de estos dos lenguajes, teniendo en cuenta la inicialización de las lambdas, no será muy conflictiva. El lenguaje más deficiente fue Java, teniendo tiempos de inicialización muy grandes con respecto a los demás. Java a diferencia de NodeJS y Python debe levantar una máquina virtual y cargar el código de la aplicación en memoria, lo cual produce que demore su primera ejecución. La elección de Java para una API Rest serverless con Lambda puede ocasionar tiempos de respuesta lentos en caso de usarse en un sistema de poco o mediano tráfico. Para futuros trabajos se puede hacer hincapié en implementar optimizaciones para acotar los tiempos de inicialización de las funciones lambda, desarrolladas con Java.

De acuerdo al test de inicialización también se pudo concluir en que Java, a pesar de ser el lenguaje que peor performance tiene en cuanto a tiempos de inicialización, es el que tiene un mayor rango de mejora de una ejecución en estado frío a una en estado



caliente. La elección de este lenguaje puede estar justificada en un sistema de tráfico moderado-alto donde sea muy poco probable tener funciones lambdas en estado frío. Los rangos de mejora de NodeJS y Python son grandes porcentualmente pero poco significativos en un caso real. Debido a que los tiempos de ejecución bajan un porcentaje grande de estado frío a caliente, se debe tener en cuenta que mientras menos tarden las funciones lambda, menos tiempo será facturado por AWS. Por ende el lenguaje con menor tiempo de ejecución será el más barato. Un caso interesante para trabajos futuros podría ser investigar acerca de las distintas alternativas para mantener las lambdas en estado caliente, siendo una de ellas ofrecida por AWS pero sumamente costosa.

Además de las propuestas a futuro, indicadas en los párrafos anteriores, sería conveniente evaluar el comportamiento de otros lenguajes de programación como Go, C#, Ruby y PowerShell.

Actualmente se están desarrollando evaluaciones de los tamaños de memoria, la variación de la performance de los lenguajes de programación y la escalabilidad. Resultados preliminares de test de memoria han determinado que los tiempos de inicialización de las lambdas implementadas en NodeJS y Python no se ven afectados por el aprovisionamiento de memoria.

## Referencias

1. Cloud computing: A complete guide. IBM. <https://www.ibm.com/cloud/learn/cloud-computing-ubl>
2. M. Armbrust, et al., Above the clouds: A Berkeley view of cloud computing, In: Tech. Rep. No. UCB/EECS-2009-28, 2009.
3. Keith D. Foote: A Brief History of Cloud Computing. <https://www.dataversity.net/brief-history-cloud-computing/> (Dec. 2021)
4. E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță and A. Iosup, : Serverless is More: From PaaS to Present Cloud Computing. In: IEEE Internet Computing, vol. 22, no. 5, pp. 8-17, Sep./Oct. 2018, doi: 10.1109/MIC.2018.053681358.
5. Adzic, G., Chatley, R.: Serverless computing: economic and architectural impact. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 884-889). ACM.(2917)
6. AWS Lambda: [aws.amazon.com/es/lambda/](https://aws.amazon.com/es/lambda/)
7. Bermbach D., Karakaya A., Buchholz S.: Using Application Knowledge to Reduce Cold Starts in FaaS Services. In: SAC '20, March 30-April 3, 2020, Brno, Czech Republic (2020).
8. Castro p., Ishakian v., Muthusamy v., Slominski a.: The rise of serverless computing. In: Communications of the ACM | Dec. 2019 | VOL. 62 | NO. 12 (2019)..
9. Rodríguez N., Atencio H. et al: Interoperabilidad de funciones en el Modelo de Programación de Serverless Computing. In: IV CICCASI. Universidad Champagnat (2020).
10. EMR: Global Serverless Computing Market Outlook
11. <https://www.expertmarketresearch.com/reports/serverless-computing-market> (2021).
12. Ping-Min Lin, Alex Glikson : Mitigating Cold Starts In Serverless Platforms A Pool-Based Approach. <https://arxiv.org/pdf/1903.12221.pdf>
13. SAND: Towards high-performance serverless computing. In Proceedings of the 2018 USENIX Annual Technical Conference. 923-935.

14. 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 15th European Conference on Computer Systems. 1-15.
15. Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with serverless-optimized containers. In Proceedings of the 2018 USENIX Annual Technical Conference. USENIX Association, 57-70.
16. Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In Proceedings of the Fourteenth EuroSys Conference 2019. 1-16.
17. P. Vahidinia, B. Farahani and F. S. Aliee: Cold Start in Serverless Computing: Current Trends and Mitigation Strategies, In: 2020 International Conference on Omni-layer Intelligent Systems (COINS), Barcelona, Spain, 2020, pp. 1-7, doi: 10.1109/COINS49042.2020.9191377.
18. Pawel Zuk and Krzysztof Rządca. 2020: Scheduling methods to reduce response latency of function as a service. In Proceedings of the 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing. 132-140
19. Artillery (s.f.) Artillery Docs [Online]. <https://www.artillery.io/docs/guides/getting-started/core-concepts>