**ISSN:** 2583-9888
**DOI:** doi.org/10.51219/JAIMLD/naresh-kalimuthu/604

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 3 & Iss: 3

*Research Article*

# The Impact of Serverless Computing on IoT Application Development: A Study on Cold Start Latency and State Management Challenges

**Naresh Kalimuthu***

**Citation:** Kalimuthu N. The Impact of Serverless Computing on IoT Application Development: A Study on Cold Start Latency and State Management Challenges. *J Artif Intell Mach Learn & Data Sci 2025* 3(3), 2895-2900. DOI: doi.org/10.51219/JAIMLD/naresh-kalimuthu/604

**Received:** 02 September, 2025; **Accepted:** 18 September, 2025; **Published:** 20 September, 2025

***Corresponding author:** Naresh Kalimuthu, USA, E-mail ID: naresh.kalimuthu@gmail.com



## ABSTRACT

Serverless computing, notably Function-as-a-Service (FaaS), serves as an appropriate execution model for Internet of Things (IoT) applications because of its scalability and pay-as-you-go model. Paradigm shifts like this come with their own sets of challenges. This paper examines serverless computing's IoT application development impact on "cold start" and state management as primary challenges and considers their impacts. The analysis documents performance degradation suffered by ergonomically IoT-sensitive systems and correlates that suffering with the scale-to-zero principle of serverless economics. In addition, the inherent statelessness of FaaS complicates the design of stateful IoT applications. The focus of this research is on mitigation techniques and architectural patterns used to address the challenges and real-world case studies are analyzed to demonstrate that the model is effective with careful implementation.

**Keywords:** Serverless Computing, Function-as-a-Service (FaaS), Internet of Things (IoT), Cold Start Latency, State Management, Cloud Computing, Edge Computing, Microservices

## 1. Introduction

### A. The serverless paradigm: A shift in cloud execution models

Serverless computing marks a significant advancement in the development of cloud execution models. Centered around the FaaS model, applications are broken into smaller, independent units called functions, which the cloud provider activates in response to designated triggers. Each paradigm that contributed to the rise of serverless computing possesses certain key features, with IaaS and PaaS offerings being the most similar to traditional models.

First, execution is fundamentally event-driven. Wrapping functions are triggered by various types of events across the entire system. These include an HTTP request via an API gateway, a database update, a queue message or a data point received from an Internet of Things (IoT) sensor. Second, these functions emphasize stateless and ephemeral processing. They run in short-lived, isolated containers or micro virtual machines and are expected to be stateless and short-lived between functions. Third, the platform guarantees automatic and seamless scaling within the system. Multiple events arriving simultaneously lead to automatic adjustments with no external control-from zero to thousands of concurrent function instances and vice versa. Finally, this model is supported by a very fine grained, pay-per-use billing system. Users are charged during function execution based on the approximate milliseconds of memory and computing time, with no costs for inactive resources. This "deliver on demand, never pay for idle" policy is the primary economic reason for its widespread adoption.

### B. Architectural requirements of the internet of things (IoT)

The Internet of Things continues to be characterized by the widespread and increasingly complex network of connected

devices, currently estimated at around 75 billion and expected to grow significantly. The devices forming the ecosystem produce vast amounts of raw data, which must be managed through advanced, flexible and scalable computational architectures. The distinctive workload patterns of data and computation shape the architectural standards of IoT applications.

The connectivity and use of IoT devices are often examined in terms of traffic. Devices can transmit data at irregular intervals, for example, when a sensor triggers an alarm or a smart device uploads data periodically. This results in sparse and random data stream traffic, which doesn't align with server-focused systems. IoT systems that play a significant role typically have low internal latency. Critical IoT applications such as industrial automation, transportation systems and monitoring devices require real-time data processing and immediate responses. This is vital for their effectiveness and security. The architecture of IoT devices must support and manage data inflow from billions of devices simultaneously. This is crucial for IoT, as the architecture needs to be easily and cost-effectively adaptable to meet constant changes in demand.

### C. The synergy and conflict of serverless and IoT

Initially, the serverless server paradigm appears well suited for IoT needs. The function-as-a-service (FaaS) model's event-driven approach matches perfectly with the event-based data generation of IoT devices and the auto-scaling capabilities can easily manage bursty and unpredictable data flows. The system's economic value is increased by the organization's payments for computation, which handles sporadic IoT events and by avoiding the costs associated with dormant servers.

A closer investigation shows that serverless computing and some specific IoT applications face a set of underlying conflicts. These conflicts stem from two distinct architectural disconnects. First, the 'scale-to-zero' feature of a serverless cost model is a primary reason why serverless computing suffers from the so-called 'cold start' problem. With this problem, function calls triggered after a period of dormancy experience high latencies. These high latencies are highly unpredictable and, as a result, are not suitable for real-time IoT applications. Furthermore, the stateless architecture and design of 'Function-as-a-Service' (FaaS) functions conflict with the needs of IoT applications, which often require tracking devices, user sessions and historical data. Consequently, application developers are forced to use inefficient systems for state control and management. This contradicts the simplicity promised by serverless computing. The stark realities of these issues are the focus of this paper, especially regarding their impact on IoT application development. The paper then explores and analyzes existing architectural frameworks and patterns aimed at easing the practical challenges faced by IoT and serverless computing.

## 2. Core Research Challenges in Serverless IoT Architectures

### A. Cold start delays in latency-sensitive IoT workflows

A cold start and any other form of latency are two distinct concepts. Cold start refers to the latency in cloud computing that occurs the first time a serverless function is triggered or when a function has not been used for a period. This is a normal occurrence. Essentially, it relates to the ability to "scale to zero," a crucial feature for the serverless business model. When a task arrives and no warm execution environment is available, the platform must perform several time-consuming steps to prepare the serverless function for execution: reserving a new container, fetching the function and its buffers, configuring the language runtime and finally executing the function. This process can be time-consuming. Research shows that cold start latency, depending on the specific functions running in a microservices environment, can range from a few milliseconds to several seconds. This can be much greater than the actual function's execution time. In complex designs with multiple function executions, this cold start issue can cause extended total response times, which in extreme cases may account for more than 90% of the total response time.

This creates a paradox for serverless IoT. The most valuable aspect of IoT serverless computing-paying for active computation on sporadic data—comes with the cost of Scaling to Zero. However, sporadic IoT traffic guarantees a sufficient load to always be active, idle or transitioning between the two. Therefore, for IoT offerings, this feature of serverless architectures is the most cost-effective, but cold starts are a significant disadvantage, often called a "massive pay-for-what-you-get" syndrome. For some IoT applications, the additional latency, which can be unpredictable and sometimes substantial, renders the system ineffective. Smart health monitoring, connected vehicles and Industrial Process Control Systems suffer greatly from this latency. This paradox presents challenges in efficient resource allocation, as the conflicting economic and performance characteristics of the serverless model complicate real-world IoT deployments.

### B. The state management dilemma for stateful IoT systems

Storage and computing function platforms rely on the assumption of statelessness, where each function call is treated as an independent, isolated transaction and executed as if previous calls did not occur. This design choice makes it easier for providers to manage resources and scale in the cloud. However, the most important IoT applications are inherently stateful. These applications need to monitor the state of a device (for example, the temperature setting on a thermostat or the lock status on a door), retain user session context (such as smart home configurations) or analyze data streams based on historical data.

This architectural disparity requires developers to export the entire application state to an external persistence layer, which could be a NoSQL database, an object store or a distributed cache. While this approach is pragmatic, it introduces significant architectural complexity and performance issues. Every function call that involves reading or writing state results in an external service network round-trip, increasing latency and negatively affecting the critical path. Additionally, developers must handle the scaling, cost and security of the external state store, which partly contradicts the 'zero-ops' promise of serverless computing.

State application centralization remains integrated within an external data monolith and is divided within the compute architecture into separate microservices. Although the microservices framework improves network latency and reduces single points of failure, the pivot functionalities reorganize the performance bloat. Adding serverless functionalities at the network edge further increases the complexity of the challenge. Among unresolved issues are maintaining invariant state across multiple self-governing, loosely connected edge

nodes containing mobile IoT devices, ensuring consistent data duplication and managing relocated state.

### C. Trade-off between performance and cost

On commercial serverless platforms, resource allocation is managed through a single dial - memory settings. The amount of allocated memory determines the CPU and network bandwidth assigned to the function. This creates a complex optimization challenge for developers, often called right-sizing the function. Under-provisioned memory can lead to longer execution times due to CPU starvation, which paradoxically increases costs if the extended duration exceeds the cheaper per-millisecond rate. Conversely, over-provisioned memory is a sunk cost, since the function will consume unnecessary additional CPU power. This forces developers to choose between balancing cost and performance for each function within an application or setting constraints.

This extends to strategic decisions at a higher level of the architecture, especially regarding function size. Developers can either create an application as many small, single-purpose functions or combine related tasks into fewer, larger functions.

- **Single-purpose functions:** This approach follows microservices principles, offering greater modularity, easier maintenance and independent scaling. However, it can increase end-to-end latency due to network delays between function calls and the risk of cascading cold starts. It may also result in direct costs for orchestration service state transitions, such as AWS Step Functions.
- **Function fusion:** Lowering inter-function latency and orchestration costs can be achieved by combining multiple steps into a single function. This approach sacrifices modularity and can lead to inefficient resource allocation because the fused function needs to reserve the most memory and CPU for the most computationally intensive task, even when handling less demanding ones.

In the case of large-scale IoT deployments, such resource management choices become even more critical. Within the context of a memory function, a slight efficiency loss or subpar architectural decision can lead to significant and unexpected cost overruns when scaled for a billion monthly invocations for a fleet of IoT devices.

## 3. Mitigation Strategies and Architectural Recommendations

### A. Taming cold start latency

Addressing the cold start problem requires using both platform-level features and disciplined application-level optimizations. Strategies can be categorized into those that aim to eliminate cold starts and those that aim to reduce the duration of unavoidable cold starts.

#### a) Platform level and proactive techniques (Reducing the Frequency)

- **Provisioned Concurrency:** This is a feature offered by cloud providers where a specified number of function execution environments are kept in a warm state and are therefore pre-initialized. Even at very low warm reachability levels, these warm instances can handle a small subset of requests. This eliminates cold starts for a predictable volume of traffic. While most cloud providers offer this at a cost and it is the most reliable method for reducing cold start latency, the resources are paid for even when not in use, which goes against the goal of minimizing operating costs through a pay-per-use model.
- **Function warming (pinging):** This is a developer-implemented strategy that uses a scheduled task to call a function at fixed intervals (every 5 minutes or more) to keep the function warm. This approach is effective for thermal pinging and is therefore limited by other deployed pods in the environment. Due to the dynamics of platform behavior, acting this way can still result in losing funds; thus, it is not guaranteed, even though it is a more cost-effective alternative to provisioned concurrency.
- **Predictive pre-warming:** These more advanced systems use machine learning to analyze historically captured invocation patterns to forecast traffic. By predicting function demand, systems can efficiently warm the necessary number of function instances just in time for usage. This approach aims to reduce the frequency of cold starts and the costs associated with unneeded warm function instances.

#### b) Application-level optimizations (Reducing Duration)

- **Code and dependency management:** A function package is a key reason for initiating a cold start, as it requires uploading, downloading and unzipping a deployment package to initialize the function. This time can be reduced by minimizing dependencies (frameworks and libraries) and using tree-shaking techniques to remove unnecessary code. Techniques like FaaSLight show that, in some cases, only the code needed for a specific invocation path can be loaded, which can decrease code loading latency by up to 79%.
- **Runtime selection:** Initialization duration is affected by the time needed to process the chosen programming language. Within interpreted languages, faster options include Python and Node, while heavier compilers like Java and .NET, dominated by the Java Virtual Machine (JVM) and Common Language Runtime (CLR), tend to have longer cold start times.
- **Container reuse and snapshotting:** Startup latency improves through FaaS platforms that reuse invoked containers. An example of a more sophisticated method of environment snapshotting is used by AWS Lambda SnapStart for Java. This method captures an environment snapshot, considering the fully initialized execution environment, the loaded JVM and the application code. The snapshot can be resumed within milliseconds on new instances, reducing initiation time compared to a cold start **(Table 1)**.

### B. Engineering stateful applications on a stateless substrate

Achieving state within an inherently stateless FaaS environment requires specific methodological frameworks for externalizing state with minimal performance impact.

**a) External persistence patterns:** In most cases, the preferred approach is to offload some of the state to an external service that is both readily available and highly scalable.

- **Databases:** Managed NoSQL databases like Amazon DynamoDB are very popular for storing and retrieving

device or session state. Their low-latency key-value access patterns match well with the request-response nature of FaaS functions, offering both durable and scalable state storage.

- **Object storage:** The primary storage layer for large unstructured data such as logs, firmware updates or media files from IoT devices is object storage services like Amazon S3. For serverless functions, there's the ability to trigger them directly with S3 events (for example, when a new file is added) and then the data is processed.
- **Distributed caches:** For a frequently accessed and time-sensitive state, placing an in-memory distributed cache (such as Redis or Memcached) in front of a primary database can reduce latency. This setup decreases the load on the database and enables access to sensitive data in approximately sub-millisecond time.

**Table 1:** Comparison of Cold Start Mitigation Techniques.

| Technique | Goal | **Advantages** | **Trade-offs** |
|---|---|---|---|
| Provisioned Concurrency | Reduce Frequency | eliminates cold starts for configured capacity. | Higher cost (billed for idle time |
| Function Warming (Pinging) | Reduce Frequency | Low cost; simple to implement. | Not guaranteed; platform may reclaim resources |
| Predictive Pre-warming | Reduce Frequency | Balances cost and performance; adapts to traffic patterns. | High complexity; requires historical data |
| Code Optimization | Reduce Duration | Reduces cost and latency for all cold starts; good practice. | Requires developer discipline |
| Runtime Selection | Reduce Duration | Simple decision with significant impact on startup time. | May conflict with team skills. |
| Environment Snapshotting | Reduce Duration | Drastically reduces initialization time for supported runtimes (e.g., Java). | Limited to specific runtimes |

**b) Using state machines for workflow orchestration:** For complex IoT processes that span multiple functions and require several tiers to maintain state, such as in multi-stage device provisioning or distributed transactions, a state machine offers an effective solution. The expressive CRUD workflow diagramming features of services like AWS Step Functions enable users to design complex workflows as visual state machines. AWS Step Functions not only manage state but also handle the invocation of Lambda functions at each state, perform error handling related to states and oversee retries associated with states. As a result, the Saga pattern is implemented, serving as the primary method for maintaining data consistency across services in a distributed system without using traditional database locks **(Figure 1)**.

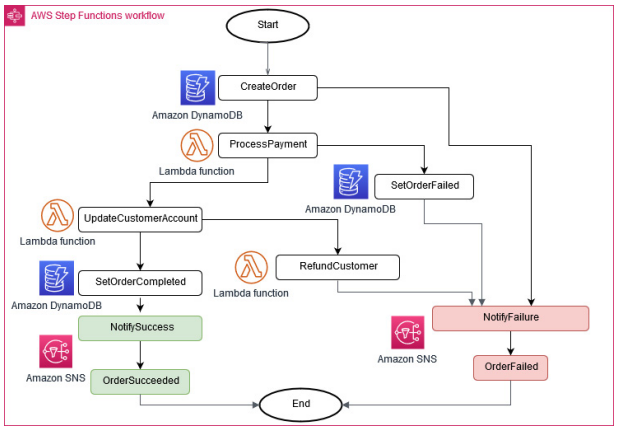


**Figure 1:** The Illustration Shows How the Saga Pattern Implements an Order Processing System by Using Aws Step Function.

**c) Stateful serverless frameworks:**New research concentrates on integrating state more naturally into the serverless model rather than merely externalizing it. The Serverless State Management Systems (SSMS) class of systems aims to reduce the challenges of stateful scaling and fault tolerance. These systems frequently use an actor-like programming model where developers create stateful message handlers and the platform manages the actor state using persistent storage checkpoints for recovery.

Numerous frameworks reflect this trend, including Cloudburst, which first introduced the principle of logical disaggregation with physical colocation. It uses a persistent, scalable and auto-scaling key-value store, while co-locating mutable caches with function executors for low-latency access to frequently used data.

Similarly, Crucial is based on the insight that FaaS involves concurrent programming at a data center scale and uses a distributed shared memory for detailed state management and synchronization. This allows for easy migration of traditional multi-threaded systems to serverless architectures with minimal code changes. Other frameworks, such as Enoki and Faasm, focus directly on stateful serverless challenges at the network edge, addressing issues related to client mobility and resource scarcity in distributed IoT systems.

The aforementioned frameworks represent a significant milestone in bridging the stateless gap in FaaS to meet the needs of advanced and stateful applications **(Table 2)**.

| **Pattern** | **Description** | **IoT Use Case** | **Fault Tolerance** | **Performance** |
|---|---|---|---|---|
| External Database | Functions read/write state to a managed NoSQL/SQL database. | Storing device shadow, user profiles, sensor readings. | High (relies on database durability and availability) | Moderate (adds network latency for every state access). |
| Distributed Cache | State is stored in an in-memory cache for fast access. | Caching frequently accessed device configurations or session data. | Moderate (cache is ephemeral; requires a backing store). | Low (sub-millisecond latency for cache hits). |
| State Machine Orchestration | AWS Step Functions manages workflow state and coordinate's function calls. | Multi-step device onboarding, complex command-and-control sequences. | Very High (built-in retries, error handling and state persistence). | Higher (overhead from the orchestration engine). |
| Stateful Frameworks | The platform provides a native state management layer (e.g., shared memory). | Real-time collaborative applications, fine-grained synchronization. | Varies by implementation (often relies on checkpointing). | Potentially Low (avoids network calls to external services). |

**B. Achieving cost-performance optimization**

Optimization strategies try to balance the two extremes of performance and cost in serverless architectures and are applied at the level of resource distribution and system architecture.

**a) Systematic function right-sizing:**

Understanding each function's guesswork is an inefficient use of resources. Developers need to adopt a strategy for each function. Automated systems such as the AWS Lambda Power Tuner can execute a function with different memory settings, analyze its performance and calculate its cost to find the optimal configuration. This strategy can be counterproductive as the ratio of memory to CPU, execution time and cost vary significantly across different workloads.

**b) Architectural guidance on function granularity:**

Choosing a combination of a large and a smaller function is a design decision that should be based on the application's use case.

- Favor functions with independent purposes (microservices) for systems that require easy segmentation into parts, independent vertical scalability and a decoupled architecture. This structure works well for loosely coupled systems that can tolerate higher costs of inter-function calls and longer delays caused by cold starts.
- When there's a sequence of tasks that are sequentially connected and network latency from calls between functions significantly impacts performance or the cost of state changes in an orchestrator becomes economically unsustainable, the appropriate approach is function fusion. This method balances a loss of modularity with gains in performance and cost efficiency, but it must be carefully designed to avoid creating monolithic functions with inefficient and unbalanced resource profiles.

## 4. Case Studies in Serverless IoT

**A) iRobot: Expanding the intelligent home**

Using a serverless architecture built on AWS Lambda and AWS IoT, iRobot successfully developed a global IoT platform for its Roomba robotic vacuums. The company began with a connected device and employed a turnkey IoT cloud provider. Still, it became clear that the service lacked the necessary scalability and extensibility for their long-term vision. This need for unlimited scalability while keeping operational costs low prompted the company to migrate to a custom serverless architecture on AWS.

This architecture serves as a perfect example of an event-driven system.

Each of the millions of robots connects to the cloud via AWS IoT Core, which functions as the primary connectivity layer. Messages from each device, such as "start cleaning," "dock," or telemetry, are received by AWS IoT Core and are in the form of device commands or telemetry data, which trigger AWS Lambda functions for further processing. This setup utilizes approximately 25 AWS services, including Amazon Kinesis for capturing streaming data in real-time and Amazon API Gateway to manage the streaming data for backend services. This model has proven to be resilient, handling unprecedented traffic volumes even 20 times higher than usual during peak times like Christmas Day, when robots are heavily utilized.

This implementation still effectively addresses the state management problem using the External Persistence Pattern. The states of each Roomba are stored in backend databases and in the AWS IoT Device Shadow service, which provides each device with a persistent virtual copy. The Lambda functions remain stateless, operating on the persistent state stored in the Shadow service.

**a) Coca-cola: The connected vending machine:** Coca-Cola has always been known for creatively adopting new technology and its AI vending machine was no different. Using serverless architecture, it was possible to build an integrated AI IoT serverless platform to support the freestyle beverage vending machines.

Coca-Cola differentiates itself with the Freestyle vending machine by allowing users to use their mobile phones to place orders for their preferred beverages. Pour by Phone was a touchless experience developed by Coca-Cola in a short period. This feature was an enhancement to the Freestyle's innovative self-serve kiosks. The mobile phones act as the control interface for the machines and real-time communication is enabled through the mobile Web 3.0 architecture deployed serverlessly. As a result, users can now place beverage orders via their mobile devices and the vending machine instantly begins pouring.

Real-time order processing is enabled by sub-360ms communication relayed through AWS. The system can achieve a 360ms turnaround time with an average SLA of 0.15 seconds. Function as a service, which supports third-party AI systems, handles complex business transactions and other actions within the agreed timeframe.

This is another great example within the global sphere for cutting costs.

Coca-Cola reduced its estimated annual operational expenses per machine from nearly $13,000 to $4,500-an impressive 66% reduction-when it shifted from a traditional to a serverless infrastructure model. The boost in operational agility proved to be even more remarkable. The company went from having just an initial idea for a touchless mobile application to deploying it on 10,000 machines in a record 100 days. This highlights the potential to accelerate the timeline for bringing complex IoT solutions to market.

## 5. Conclusion

This research demonstrates that while serverless computing is an effective approach for developing and innovating scalable, low-cost IoT applications, some technical issues still need to be addressed. These issues stem from the core features of serverless computing: ephemeral and stateless functions based on the scale-to-zero concept. Such functions can lead to performance problems like cold start latency and complex state management. The paper shows how cold starts and latency-sensitive IoT systems become vulnerable and explains how modern FaaS architectures must make compromises to overcome the limitations of statelessness, making these problems more complex. However, these challenges are manageable. Developers can effectively handle them using platform-level features like Provisioned Concurrency, application-level optimizations such as disciplined code right-sizing and proven architectural solutions including external state persistence and state machine orchestration. Success stories from industry leaders like iRobot and Coca-Cola confirm this. These

examples strongly indicate that, if these challenges are properly managed, the serverless model can deliver on its promise as an effective, revolutionary and powerful foundation for the next generation of IoT applications.

## 6. References


1. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html
2. https://dl.acm.org/doi/10.1145/3352460.3358296
3. https://arxiv.org/abs/1902.03383
4. https://link.springer.com/chapter/10.1007/978-981-10-5026-8_1
5. https://arxiv.org/abs/2208.04213
6. https://arxiv.org/abs/2310.08437
7. https://dl.acm.org/doi/10.1145/3406011