A Comparative Study of Cold Start Mitigation Techniques in Modern Function-as-a-Service Architectures

Prathamesh Mohan Lohar[1] Mrs. Ashwini R. Garkhedkar[2]
[1]Student [2]Assistant professor
[1,2]Master of Computer Applications
[1,2]PES's Modern College of Engineering, Pune, Maharashtra, India

Abstract — Serverless computing has emerged as one of the most influential paradigms in cloud-native application development due to its elasticity, operational simplicity, and event-driven billing model. By abstracting infrastructure management from developers, Function-as-a-Service (FaaS) platforms enable rapid deployment and automatic scaling while minimizing operational overhead. Despite these advantages, cold start latency remains one of the most significant technical barriers preventing broader adoption in latency-sensitive applications. Cold starts occur when a serverless platform must initialize a new execution environment before serving a request, introducing delays caused by infrastructure provisioning, runtime startup, dependency loading, and application bootstrap logic. While historically treated primarily as a performance concern, cold starts increasingly represent an architectural and economic challenge in burst-driven workloads. This paper presents a comparative architectural analysis of cold start behavior across modern serverless runtimes, focusing on Node.js, Python, and Java. The study evaluates runtime initialization mechanisms, dependency management patterns, packaging strategies, snapshot-based optimization approaches, and infrastructure-level tuning techniques. Additionally, this work introduces the Serverless Efficiency Index (SEI), a decision framework designed to help practitioners evaluate optimization strategies by balancing latency reduction, operational cost, engineering effort, and implementation risk. The findings suggest that effective cold start mitigation requires a multi-dimensional approach spanning application architecture, runtime selection, packaging discipline, and infrastructure orchestration.

Keywords: Serverless Computing, Function-as-a-Service, Cold Start, AWS Lambda, Node.js, Python, Java, SnapStart, GraalVM, Cloud Optimization

I. Introduction

Cloud computing has evolved significantly from monolithic infrastructure provisioning to highly abstracted execution paradigms. Serverless computing represents one of the most prominent shifts in this evolution, enabling developers to deploy event-driven code without explicit infrastructure management. In the serverless model, the cloud provider assumes responsibility for provisioning, scaling, fault tolerance, and runtime maintenance, allowing development teams to focus solely on business logic.

Function-as-a-Service (FaaS) platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions have accelerated serverless adoption due to their operational simplicity and consumption-based pricing model. These systems automatically scale function instances in response to demand while billing users only for execution time and allocated resources.

Despite these advantages, serverless architecture introduces a fundamental performance challenge: cold starts.

A cold start occurs when a request arrives for a function without an available warm execution environment. The platform must therefore provision infrastructure, initialize the language runtime, load application dependencies, and execute startup logic before processing the request. This initialization delay can range from milliseconds to several seconds depending on runtime characteristics, dependency complexity, and infrastructure conditions.

For workloads with strict latency requirements—including financial transactions, interactive APIs, authentication services, gaming backends, and IoT control systems—cold start latency can significantly degrade user experience and violate service-level objectives.

Cold start behavior is influenced by multiple dimensions:

1) Runtime architecture
2) Dependency footprint
3) Packaging strategy
4) Infrastructure allocation
5) Provisioning model
6) Application framework complexity

The severity of the problem varies significantly between runtimes. Lightweight interpreted environments such as Python typically exhibit shorter initialization times for small applications, whereas Java-based functions historically suffer from substantial startup overhead due to JVM initialization and class loading. Modern innovations such as snapshot restoration and ahead-of-time compilation have begun to reshape these trade-offs.

This paper investigates cold start behavior across major runtimes and evaluates optimization strategies from both performance and architectural perspectives.

The primary contributions of this paper are:

1) A systematic architectural breakdown of serverless cold start phases.
2) Comparative runtime analysis of Node.js, Python, and Java.
3) Evaluation of modern mitigation techniques including bundling, lazy loading, snapshot restoration, and native compilation.
4) Infrastructure-level analysis including memory tuning and concurrency provisioning.
5) Proposal of the Serverless Efficiency Index (SEI) for optimization decision-making.

II. Background: Anatomy of a Cold Start

Cold starts are not singular events but composite initialization workflows involving multiple subsystems.

A serverless cold start can generally be divided into three primary phases:

1) Infrastructure Provisioning
2) Runtime Initialization

3) Application Initialization

A. Infrastructure Provisioning

Infrastructure provisioning represents the platform-managed initialization layer.

During this phase, the provider allocates computational resources necessary to host the function execution environment. Depending on platform implementation, this may involve container startup, micro-VM allocation, filesystem preparation, and network attachment.

Typical infrastructure preparation tasks include:

- CPU allocation
- Memory reservation
- Storage mounting
- Network interface configuration
- Security isolation setup
- Runtime image retrieval

Modern platforms increasingly use lightweight virtualization technologies to reduce provisioning overhead. For example, micro-VM-based architectures provide strong isolation while minimizing startup delay compared to traditional virtual machines.

Although infrastructure provisioning is largely opaque to developers, its impact remains significant in burst-scaling scenarios.

B. Runtime Initialization

Runtime initialization begins after infrastructure allocation completes.

This phase depends heavily on language runtime design.

1) Node.js Runtime Behavior:

Node.js relies on the V8 JavaScript engine. Startup requires initialization of the JavaScript execution environment, internal libraries, event loop infrastructure, and parsing mechanisms.

The startup path includes:

- V8 engine startup
- Heap allocation
- Context creation
- Module resolution initialization
- Event loop setup

Although relatively lightweight compared to JVM startup, runtime initialization still contributes measurable latency.

2) Python Runtime Behavior

Python serverless functions typically use CPython.

Initialization includes:

- Interpreter startup
- Module loader initialization
- Built-in library loading
- Environment configuration
- Site package preparation

Python startup remains efficient for minimal applications but becomes sensitive to dependency volume.

3) Java Runtime Behavior

Java exhibits the most complex runtime initialization.

Startup typically includes:

- JVM bootstrap
- Heap initialization
- Garbage collector startup
- Class loader preparation
- Bytecode verification
- Reflection metadata preparation
- JIT compiler initialization

Historically, this startup complexity has made Java less favorable for traditional serverless workloads.

C. Application Initialization

Application initialization includes user-controlled startup logic.

Typical activities include:

- Framework bootstrap
- SDK loading
- Dependency imports
- Configuration parsing
- Environment variable loading
- Secret retrieval
- Database connection setup
- Logging initialization

Unlike infrastructure provisioning, this phase is directly influenced by application design decisions.

In many practical workloads, application initialization dominates cold start duration.

III. Related Work

Cold start optimization has received significant attention in both academic and industrial research.

Early serverless studies focused primarily on measuring latency behavior across cloud providers and identifying runtime startup variability. These works established that language runtime choice significantly impacts initialization performance.

Subsequent research explored mitigation strategies including container reuse, execution pre-warming, snapshot restoration, ahead-of-time compilation, and adaptive scheduling.

Snapshot-based optimization has emerged as a major innovation in reducing JVM startup costs, while native compilation approaches have demonstrated promising startup improvements at the expense of compatibility complexity.

Infrastructure-aware optimization has also gained relevance, particularly regarding memory-to-CPU proportionality and concurrency provisioning.

Despite extensive prior work, a practical decision-oriented framework combining runtime characteristics, infrastructure tuning, and economic trade-offs remains limited.

This paper contributes by synthesizing these dimensions into a unified architectural evaluation framework.

IV. Comparative Runtime Analysis

Runtime selection remains one of the most influential decisions affecting serverless cold start behavior. While infrastructure provisioning introduces baseline latency, runtime architecture determines startup complexity, dependency execution cost, memory initialization overhead, and optimization opportunities.

This section evaluates Node.js, Python, and Java from both performance and architectural perspectives.

V. Node.js Runtime Analysis

Node.js has become one of the most widely adopted runtimes for serverless workloads due to its event-driven execution model, lightweight startup behavior, and strong ecosystem support.

Its relatively fast cold start characteristics make it attractive for API endpoints, webhooks, authentication handlers, lightweight orchestration, and event-driven integration services.

However, practical performance depends heavily on dependency discipline and packaging strategy.

A. V8 Engine Architecture

Node.js executes JavaScript using Google's V8 engine.
Unlike ahead-of-time compiled binaries, JavaScript must be parsed and transformed into executable representations during startup.
The V8 execution pipeline typically involves:

1) Source parsing
2) Abstract syntax tree generation
3) Bytecode generation
4) Immediate interpretation
5) JIT optimization for hot code paths

During cold starts, the primary overhead occurs before the handler executes.
Key initialization components include:

- V8 engine startup
- Memory heap allocation
- Execution context creation
- Event loop initialization
- Module resolution preparation

Although relatively lightweight compared to JVM startup, these operations become significant in dependency-heavy applications.

B. JIT Compilation Overhead

V8 employs a staged optimization model.
Initially, JavaScript executes through an interpreter layer optimized for startup responsiveness. Frequently executed code paths may later be compiled into optimized machine code.

In serverless workloads, however, many functions execute for short durations and terminate before substantial optimization occurs.
This creates a trade-off:

- Fast startup interpretation
- Limited opportunity for deep optimization

As a result, cold start-sensitive Node.js workloads benefit more from startup reduction than runtime optimization.

C. Dependency Explosion Problem

Node.js applications frequently suffer from dependency inflation.

The ecosystem encourages modular package composition, which improves development velocity but often results in extremely large dependency graphs.
A single business-level dependency may transitively import hundreds of subpackages.
Consequences include:

- Increased filesystem traversal
- More module resolution work
- Additional parsing overhead
- Higher memory consumption
- Longer startup initialization

The issue is amplified in serverless environments where each cold start re-executes initialization logic.

D. CommonJS vs ECMAScript Modules

Module architecture significantly affects optimization opportunities.
Traditional CommonJS uses dynamic loading:

```
const sdk = require('library');
```

Dynamic imports reduce static predictability.
Modern ECMAScript Modules (ESM) provide explicit import declarations:

```
import { fn } from 'library';
```

Benefits include:

- Better tree shaking
- Reduced dead code inclusion
- Improved bundling optimization
- Lower deployment artifact size

Static dependency analysis substantially improves packaging efficiency.

E. Bundling and Minification

Bundling remains one of the most effective Node.js cold start optimizations.
Modern tools include:

- esbuild
- Rollup
- Webpack
- SWC

Benefits:

- Fewer filesystem calls
- Reduced module resolution
- Smaller deployment packages
- Faster initialization

Best practice includes:

- single-entry bundle generation
- dependency pruning
- SDK modularization
- dead code elimination

F. Node.js Optimization Strategies

Practical recommendations:

1) Use bundling aggressively
2) Prefer modular SDKs
3) Avoid heavyweight middleware chains
4) Lazy-load rarely used libraries
5) Minimize startup framework initialization
6) Reduce deployment artifact size

VI. Python Runtime Analysis

Python remains highly popular in serverless computing due to simplicity, ecosystem maturity, readability, and extensive cloud tooling.

Python performs well in lightweight event-driven applications but becomes sensitive to dependency-heavy initialization.

A. CPython Startup Characteristics

Python functions generally rely on CPython.
Initialization includes:
- interpreter startup
- import system initialization
- built-in module preparation
- environment setup
- package resolution

Minimal Python functions exhibit fast startup behavior.
However, practical applications frequently suffer from dependency inflation.

B. Import Overhead

Python imports execute module-level code.
This differs from simple symbol resolution.
When importing a package:
- module files are located
- bytecode may be compiled
- initialization code executes
- transitive dependencies load

For large frameworks, startup cascades can become substantial.
Examples of expensive imports:
- pandas
- numpy
- sqlalchemy
- boto3
- machine learning libraries

Cold start performance therefore depends heavily on dependency hygiene.

C. Dependency Amplification

Python ecosystems often encourage convenience over startup efficiency.
Common problems:
- unnecessary framework imports
- ORM-heavy architectures
- large serialization libraries
- unused scientific packages

Even moderate application complexity may significantly increase initialization latency.

D. Lazy Importing

Lazy loading defers expensive initialization.
Example:
Instead of:

```
import boto3
```

Use:

```
def handler(event, context):
    import boto3
```

Benefits:
- reduced cold start baseline
- path-specific optimization
- conditional dependency loading

Trade-off:
First execution of lazy code incurs delay.
Therefore suitability depends on workload behavior.

E. Lightweight Dependency Alternatives

Dependency substitution often yields substantial gains.
Examples:
- native DB drivers instead of heavy ORMs
- minimal JSON parsers
- targeted utility packages
- direct HTTP clients

Engineering discipline strongly affects Python cold start performance.

F. Python Optimization Strategies

Recommended practices:
1) minimize dependency graph size
2) lazy-load conditional functionality
3) avoid heavyweight frameworks
4) reduce package artifact size
5) replace convenience-heavy libraries

VII. Node.js vs Python Comparative Analysis

Node.js and Python both perform well for serverless workloads but differ in scaling behavior.

A. Runtime Behavior Comparison

Characteristic	Node.js	Python
Baseline Startup	Fast	Fast
Dependency Scaling	Better	Worse
Packaging Tooling	Strong	Moderate
Bundling Efficiency	High	Lower
Import Overhead	Moderate	High
Developer Simplicity	Moderate	High

Node.js generally scales better in dependency-heavy production applications due to mature bundling ecosystems.
Python remains highly effective for lightweight workloads.

VIII. Java Runtime Analysis

Java has historically been considered problematic for serverless cold start-sensitive workloads.

This perception originated from JVM startup complexity.
However, recent innovations have substantially altered this assessment.

A. JVM Initialization Cost

JVM startup includes:
- heap allocation
- metaspace initialization
- garbage collector startup
- class loader creation
- bytecode verification
- reflection metadata preparation
- JIT subsystem initialization

These operations create significant cold start overhead.

B. Framework Amplification

Enterprise Java frameworks often increase startup cost.
Examples:
- Spring Boot

– Jakarta EE ecosystems
– reflection-heavy dependency injection
– annotation scanning

Framework startup often dominates initialization latency.

C. Traditional Trade-off

Historically:

– poor startup latency
– excellent warm throughput
– strong enterprise tooling
– mature ecosystem support

This created a startup-versus-throughput trade-off.

D. Modern Relevance

Snapshot restoration and native compilation have significantly improved Java's viability in serverless environments.

This evolution is analyzed in the next section.

IX. Snapshot-Based Runtime Optimization

One of the most significant developments in serverless runtime optimization has been the emergence of snapshot-based execution restoration.

Rather than repeatedly executing expensive initialization sequences for each cold start, snapshot restoration captures a pre-initialized runtime state and restores it when needed.

This fundamentally changes the startup economics of traditionally heavy runtimes.

X. AWS SnapStart and Coordinated Restore at Checkpoint (CRaC)

Snapshot restoration has become especially relevant for Java workloads.

Traditional JVM startup overhead historically limited Java adoption in cold start-sensitive serverless systems.

Snapshot-based optimization significantly reduces this limitation.

A. Conceptual Model

Snapshot optimization follows a checkpoint-and-restore lifecycle:

1) Deploy application
2) Initialize runtime fully
3) Execute startup logic
4) Capture runtime memory snapshot
5) Store checkpoint image
6) Restore snapshot on invocation

Instead of rebuilding execution state repeatedly, the system restores previously initialized memory state.

This converts initialization-heavy startup into a restore operation.

B. Architectural Benefits

Advantages include:

– elimination of repeated JVM bootstrap
– avoidance of repeated class verification
– removal of repeated framework startup
– faster application readiness
– improved consistency in startup behavior

This transforms Java's practical startup profile.

C. CRaC Lifecycle Hooks

Coordinated Restore at Checkpoint (CRaC) introduces lifecycle control around snapshot events.

Typical lifecycle events include:

– before checkpoint
– after restore

These hooks allow applications to control state preparation.

Common use cases:

– close network connections
– flush caches
– initialize frequently used objects
– pre-load expensive resources
– re-establish volatile runtime state

This provides practical control over restore correctness.

D. Priming Strategies

Snapshot effectiveness depends on what state exists before checkpoint creation.

Priming refers to deliberately warming execution state prior to snapshot capture.

Examples:

– forcing class loading
– triggering reflection initialization
– executing serialization paths
– warming JSON parsers
– pre-initializing framework components

Benefits:

– reduced post-restore latency
– fewer first-request penalties
– better warm-path consistency

Well-primed snapshots can dramatically improve practical responsiveness.

XI. Operational Risks of Snapshot Restoration

Snapshot-based execution introduces architectural complexity.

A. Stale Network Connections

Connections created before checkpoint capture may become invalid after restoration.

Examples:

– database sockets
– message queue sessions
– HTTP keep-alive channels

Mitigation:

– close before checkpoint
– reconnect after restore

B. Randomness Duplication

Stateful random generators may be duplicated across restored instances.

Potential consequences:

– repeated UUIDs
– duplicate session identifiers
– cryptographic weaknesses

Randomness must be refreshed post-restore.

C. Time Synchronization

Snapshot restore may preserve stale temporal assumptions.
Examples:
– expired tokens
– time-window validation failures
– scheduler drift
Applications should refresh time-sensitive state after restoration.

D. Configuration Drift

External configuration may change after snapshot creation.
Potential issues:
– outdated credentials
– stale feature flags
– obsolete service endpoints
Architectural discipline is required.

XII. Ahead-of-Time Compilation with GraalVM Native Image

An alternative to snapshot restoration is ahead-of-time compilation.
GraalVM Native Image transforms Java applications into native executables.
This removes dependency on JVM startup.

A. Execution Model

Traditional Java:
– load JVM
– initialize heap
– verify classes
– start JIT
– execute code
Native Image:
– launch native binary
– execute directly
This significantly reduces startup latency.

B. Advantages

Benefits include:
– extremely fast startup
– reduced memory footprint
– predictable initialization
– simplified deployment runtime

C. Trade-offs

Challenges include:
– reflection configuration complexity
– dynamic feature limitations
– native build complexity
– framework compatibility constraints
Enterprise applications may require adaptation.

D. Snapshot vs Native Compilation Comparison

Characteristic	SnapStart	GraalVM Native
Startup Speed	Very Fast	Extremely Fast
Warm Throughput	High	Moderate
Compatibility	High	Lower
Developer Effort	Moderate	High
Memory Efficiency	Moderate	High
Framework Support	Strong	Variable

XIII. Infrastructure-Level Optimization

Runtime optimization alone is insufficient.
Infrastructure configuration strongly affects cold start performance.

A. Memory-to-CPU Proportionality

Many serverless platforms allocate CPU proportionally to configured memory.
Consequences:
– faster runtime startup
– improved dependency parsing
– reduced JIT initialization time
– quicker framework bootstrap
Higher memory configurations may therefore reduce total execution cost despite increased per-unit pricing.

B. Power Tuning

Power tuning involves benchmarking multiple configurations.
Typical tuning process:
1) deploy workload variants
2) measure cold start latency
3) measure execution duration
4) calculate total billed cost
5) select optimal trade-off
This prevents under-provisioned inefficiency.

XIV. Provisioned Concurrency

Provisioned concurrency eliminates cold starts by maintaining pre-initialized execution environments.

A. Benefits

Advantages:
– predictable latency
– no cold start delay
– SLA protection
– stable API responsiveness

B. Trade-offs

Costs include:
– fixed infrastructure spend
– reduced elasticity efficiency
– capacity planning complexity
Provisioned concurrency is most appropriate when latency guarantees outweigh cost sensitivity.

XV. Economic Decision Framework for Provisioned Concurrency

Provisioned concurrency should be evaluated economically.
Break-even depends on:
– cold start frequency
– cold start penalty cost
– fixed concurrency charges
– latency SLA requirements
Decision heuristics:
– bursty low-frequency workloads may tolerate cold starts
– user-facing low-latency APIs may justify pre-warming

– enterprise SLAs often favor deterministic response times

Optimization should be driven by workload economics rather than assumptions.

XVI. Serverless Efficiency Index (SEI)

Selecting a cold start optimization strategy requires balancing multiple competing factors. Startup latency reduction alone is insufficient as an optimization objective because engineering effort, maintenance complexity, operational risk, and infrastructure cost significantly influence practical architectural decisions.

To support decision-making, this paper proposes the Serverless Efficiency Index (SEI).

$$SEI = \frac{(\Delta L_{p99} \cdot \omega_1) + (\Delta C \cdot \omega_2)}{D \cdot R}$$

Where:

ΔL_{p99} = percentage reduction in tail latency

ΔC = estimated cost savings

ω_1, ω_2 = organizational weighting factors

D = engineering implementation effort

R = operational risk factor

This model encourages optimization decisions aligned with organizational priorities.

A. Interpretation

Example interpretations:

– High latency reduction + low complexity = high SEI
– Moderate performance gains + extreme implementation risk = low SEI
– Expensive optimization with minimal business impact = poor candidate

SEI is intended as a comparative decision framework rather than an absolute universal metric.

XVII. Future Research Directions

Serverless runtime evolution continues rapidly.

Several research directions appear especially promising.

A. Predictive Execution Warming

Predictive warming aims to proactively initialize execution environments before anticipated demand spikes.

Potential enabling techniques:

– time-series traffic forecasting
– adaptive workload modeling
– request-pattern clustering
– scheduler-level anticipation

If implemented effectively, predictive warming could reduce cold start frequency while preserving elasticity.

B. WebAssembly-Based Serverless Runtimes

WebAssembly (WASM) introduces lightweight execution environments with near-native performance.

Potential advantages:

– extremely fast startup
– reduced runtime overhead
– smaller deployment artifacts
– improved portability

WASM-based FaaS models may fundamentally alter runtime trade-offs.

C. Adaptive Runtime Specialization

Future platforms may dynamically optimize execution environments based on workload behavior.

Examples:

– workload-aware pre-initialization
– runtime-specific optimization policies
– hybrid snapshot/native deployment models

D. Scheduler-Aware Placement

Cold start performance may improve through intelligent placement strategies considering:

– cache locality
– regional demand patterns
– runtime reuse opportunities
– infrastructure affinity

XVIII. Threats to Validity

Several limitations affect this study.

A. Platform Variability

Cold start behavior varies across providers.

Differences include:

– container architecture
– virtualization model
– scheduler implementation
– networking design

Results should therefore be interpreted architecturally rather than as provider-specific guarantees.

B. Workload Dependency

Runtime behavior depends heavily on application characteristics.

Examples:

– dependency graph size
– framework selection
– connection initialization
– serialization complexity

Generalized comparisons cannot fully capture workload-specific variation.

C. Rapid Platform Evolution

Serverless platforms evolve rapidly.

New runtime features, pricing changes, optimization techniques, and scheduling mechanisms may alter comparative behavior over time.

D. Economic Assumptions

Cost modeling depends on provider pricing models, execution frequency, memory allocation, and concurrency behavior.

Economic conclusions should therefore be adapted to deployment-specific contexts.

XIX. Conclusion

Cold start latency remains one of the most important architectural constraints in serverless computing.

This paper examined cold start behavior across Node.js, Python, and Java runtimes from runtime, infrastructure, and operational perspectives.

The analysis shows that:

- Node.js performs strongly when dependency packaging is disciplined.
- Python remains highly effective for lightweight workloads but degrades under dependency-heavy architectures.
- Java has become substantially more competitive through snapshot restoration and native compilation.
- Infrastructure configuration significantly influences practical performance.
- Economic optimization must accompany technical optimization.

Cold start mitigation is not a single engineering technique but a systems-level architectural decision.

Effective optimization requires balancing runtime behavior, infrastructure economics, engineering effort, and operational reliability.

The proposed Serverless Efficiency Index provides a structured mechanism for evaluating these trade-offs.

Future advances in predictive warming, WebAssembly runtimes, and adaptive scheduling may significantly reduce cold start impact, but architectural discipline will remain essential.

ACKNOWLEDGMENT

The author acknowledges the broader serverless research community and publicly available cloud platform documentation that informed this architectural analysis.

REFERENCES

[1] Amazon Web Services. 2025a. AWS Lambda Developer Guide. https://docs.aws.amazon.com/lambda/.
[2] Amazon Web Services. 2025b. *AWS Lambda SnapStart Documentation*. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html.
[3] Baldini, Ioana et al. 2017. "Serverless Computing: Current Trends and Open Problems." *Research Advances in Cloud Computing*.
[4] Google Cloud. 2025. *Cloud Functions Documentation*. https://cloud.google.com/functions.
[5] Jonas, Eric, Johann Schleier-Smith, et al. 2019. "Cloud Programming Simplified: A Berkeley View on Serverless Computing." *arXiv Preprint arXiv:1902.03383*.
[6] McGrath, Gareth, and Paul Brenner. 2017. "Serverless Computing: Design, Implementation, and Performance." *IEEE Distributed Computing Systems Workshops*.
[7] Microsoft. 2025. *Azure Functions Documentation*. https://learn.microsoft.com/azure/azure-functions/.
[8] OpenJDK. 2025. *Coordinated Restore at Checkpoint (CRaC)*. https://openjdk.org/projects/crac/.
[9] Oracle. 2025. *GraalVM Native Image*. https://www.graalvm.org.
[10] Spillner, Josef. 2017. "Snafu: Function-as-a-Service Runtime Design and Execution Model." *arXiv*.
[11] Wang, Liang et al. 2018. "Peeking Behind the Curtains of Serverless Platforms." *USENIX Annual Technical Conference*.

