Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts

Sumer Kohli*
Stanford University

Shreyas Kharbanda*†
Cornell University

Rodrigo Bruno
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Joao Carreira
University of California, Berkeley

Pedro Fonseca
Purdue University

Abstract

Serverless computing allows developers to deploy and scale stateless functions in ephemeral workers easily. As a result, serverless computing has been widely used for many applications, such as computer vision, video processing, and HTML generation. However, we find that the stateless nature of serverless computing wastes many of the important benefits modern language runtimes have to offer. A notable example is the extensive profiling and Just-in-Time (JIT) compilation effort that runtimes implement to achieve acceptable performance of popular high-level languages, such as Java, JavaScript, and Python. Unfortunately, when modern language runtimes are naively adopted in serverless computing, all of these efforts are lost upon worker eviction. Checkpoint-restore methods alleviate the problem by resuming workers from snapshots taken after initialization. However, production-grade language runtimes can take up to thousands of invocations to fully optimize a single function, thus rendering naive checkpoint-restore policies ineffective.

This paper proposes Pronghorn, a snapshot serverless orchestrator that automatically monitors the function performance and decides (1) when it is the right moment to take a snapshot and (2) which snapshot to use for new workers. Pronghorn is agnostic to the underlying platform and JIT runtime, thus easing its integration into existing runtimes and worker deployment environments (container, virtual machine, etc.). On a set of representative serverless benchmarks, Pronghorn provides end-to-end median latency improvements of 37.2% across 9 out of 13 benchmarks (20-58% latency reduction) when compared to state-of-art checkpointing policies.

*Both authors contributed equally to the paper.
†Work primarily conducted while at Purdue University.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3629556

ACM Reference Format:
Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3627703.3629556

1 Introduction

Serverless computing is an emerging cloud computing paradigm that leverages lightweight, stateless workers to streamline application deployment. Under the serverless model, developers decompose their applications into lightweight, stateless logic units (*functions*) written in languages such as Python, JavaScript, and Java. In contrast with traditional cloud computing, serverless platforms automatically handle infrastructure provisioning and application scalability, enabling cloud developers to focus on their application's business logic [21, 22]. We will refer to serverless computing and function-as-a-service (FaaS) interchangeably as this paper focuses on cloud offerings that combine both aspects at the same time.

Serverless computing leverages high-level languages, freeing developers from many concerns related to managing software packages and dependencies, low-level resources, deployment, and portability. In turn, many of the benefits provided by these languages are only possible by the use of sophisticated language runtimes, such as PyPy [11] (a high-performance Python implementation) and OpenJDK HotSpot JVM [5] (the most widely used JVM implementation), which abstract programmers from the underlying hardware resources while still offering acceptable run time performance.

To achieve high application performance and ease of use, modern language runtimes rely on complex JIT compilers that incrementally gather information about an application's execution and iteratively optimize it over many invocations. This approach allows runtimes to leverage different techniques, in particular, dynamic and speculative optimizations [28, 63]. These optimizations include method inlining and devirtualization [37], branch prediction, type inference [35], loop unrolling and code hoisting [17], vectorization [52], and others. In practice, runtime optimizations

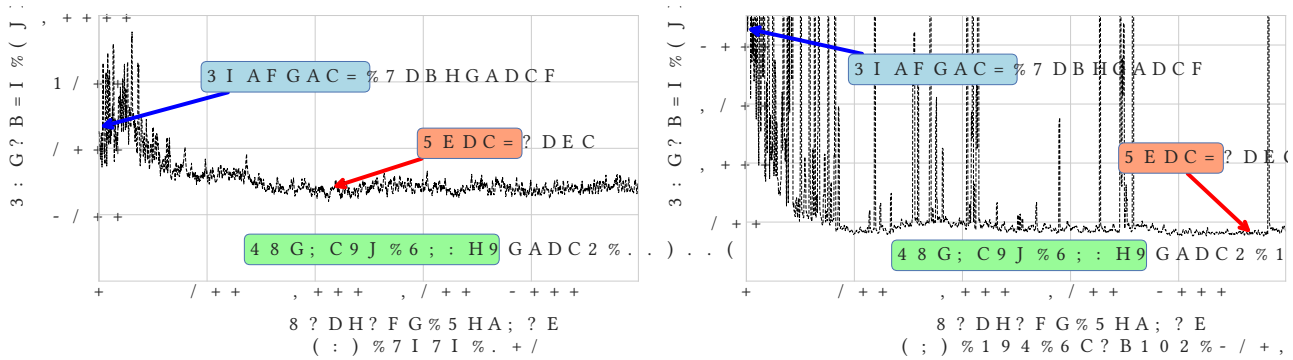

Figure 1. Dynamic HTML generation workload latency using two optimizing runtimes. Experiment shows the performance difference produced by taking a premature snapshot (as existing solutions do) and an ideal snapshot. These benchmarks were repeatedly invoked for about 2500 requests, since convergence took up to approximately that number of requests.

are key to achieving acceptable performance in high-level languages, and disabling them significantly degrades performance [23].

Unfortunately, the design of today's serverless platforms is at odds with modern language runtimes for two main reasons and effectively prevents serverless platforms from using resources efficiently. First, serverless functions are stateless, hence, serverless platforms are designed to discard function state, including the JIT state, after each invocation. Second, serverless platforms serve many function invocations across many workers, spanning multiple customers and applications. As a result, each container individually might only see a few invocations throughout its lifetime, a fundamentally inefficient scenario for mature language runtimes.

In contrast, modern language runtimes are better suited to host a single long-running application where code is continuously profiled and optimized. Figure 1 shows how a typical serverless workload (HTML page rendering), from the SeBS [24] benchmark suite, can take on the order of two thousand successive requests to converge to optimal performance, far beyond the typical serverless worker lifetime.[1] In fact, we observe the same effect for entirely different workloads and JIT runtimes (see §5), demonstrating the importance of runtime optimizations for efficiently executing high-level code in the cloud. Worse, the stateless nature of serverless implies that with every new worker, all previous runtime optimizations are lost, requiring past optimizations to be wastefully re-computed each time. This results in significant missed performance gains, as seen in previous work [23].

In order to address this challenge, researchers and practitioners have turned to checkpoint-restore approaches to retain the code optimizations generated by language runtimes. Checkpoint-restore allows serverless platforms to take a snapshot of a function's state to be later restored in order to serve a new function invocation. Checkpointing a worker (function instance) can happen at any stage of the function execution, for instance, after the code has been loaded into memory, after code compilation, or even after the function has been executed a few times. In fact, we find that the timing of when to checkpoint the state of a function is critical to achieving acceptable performance after restoring it (see §2).

Even though the decision of when to checkpoint a function can have a significant impact on the performance of the application after restoring it, no solution currently proposes a technique to decide *when* to checkpoint a function. In fact, existing solutions that use checkpoint-restore use a single predefined threshold to decide when to checkpoint [15, 27, 59, 61], typically after runtime initialization and before the first function invocation, or after the first invocation. We show that neither of these approaches is sufficient to capture runtime optimizations that maximize performance (see §2).

In this paper, we propose Pronghorn, a snapshot orchestrator that ensures that new containers start with a fully optimized version of each serverless function. Pronghorn learns the best moment to checkpoint each function by taking snapshot samples over a configurable period of time. Pronghorn targets rapid convergence and continuous learning, the latter of which provides resilience to perturbations in serverless function input (see §5). Finally, Pronghorn allows cloud providers to control the snapshot storage and network overheads incurred during the learning period, and, hence, control the cost-performance trade-off.

To show the performance benefits of Pronghorn, we evaluated it with a set of 13 representative serverless function benchmarks from the ServerlessBench [71], FaasDom [46], and SeBS [24] benchmark suites and previous papers [23], using realistic values of serverless worker eviction rates [58], function invocation frequencies, and variance in workload latencies. We find that Pronghorn can provide cost-efficient

[1] Serverless workers are typically evicted due to inactivity after a platform-specific timeout, *e.g.,* 10 minutes in AWS Lambda. Moreover, a recent study [58] revealed that only ≈25% of functions receive more than one invocation under 10 minutes, meaning that many containers will only ever see a single invocation.

	Serverless Benchmark			
	Hash	**HTML**	**WordCount**	**JSON**
Request #1 (baseline)	27 ms	650 ms	64 ms	360 ms
Request #200	1.9x	1.1x	2.3x	4.3x
Request #400	1.5x	3.9x	2.8x	**5.9x**
Request #600	**2.5x**	4.7x	**3.4x**	3.9x
Request #800	2.0x	**5.1x**	1.8x	2.6x

Table 1. Function latency reduction compared with the first request for Java benchmarks. Different benchmarks achieve peak performance at different numbers of served requests. Benchmarks were invoked for up to 1000 requests to allow the code to be extensively optimized, although full convergence may take longer. Measurements were taken on an Ubuntu 18.04 LTS VM with 4 GiB RAM and OpenJDK 17.0.1.

function invocation latency reductions of 20%-58% relative to the state-of-the-art in exchange for additional checkpoint operations that are not on the critical path (can be executed when the runtime is idle) and a configurable amount of additional storage which is used keep the snapshot pool. Overall, Pronghorn improves median end-to-end latency by 37.2% on 9 out of 13 benchmarks.

In summary, this work makes the following contributions:

- We analyze the impact of the snapshot point during function execution on function performance. This analysis quantifies the role of language runtime optimizations on serverless performance;
- We propose an effective snapshot policy that will orchestrate snapshot samples and search for a snapshot that optimizes function execution. To the best of our knowledge, this is the first attempt to automatically optimize snapshot orchestration;
- We compare our snapshot orchestrator policy against other orchestration policies used in the state-of-art. Our results show that using our proposed policy yields significant performance benefits.

2 Challenges of Serverless Checkpoint-Restore

A number of previous works leverage checkpoint-restore techniques to accelerate container and runtime initialization stages of a function invocation [20, 27, 61]. Despite their success, such works miss significant performance benefits (up to 61% of end-to-end latency in our experiments) since they do not fully exploit the optimizations provided by runtimes. In order to capture these benefits, serverless platforms need to be made aware that runtimes profile and optimize code over multiple executions. In this section, we explain the main challenges related to doing so.

Complex language runtimes. Modern language runtimes make use of highly complex optimizing pipelines that are critical to improving the performance of high-level code. For instance, applications running on OpenJDK HotSpot JVM [5] (the most widely used industrial Java Virtual Machine) are initially executed by the runtime interpreter. At this stage, the runtime has to load the code into memory, parse it and initialize internal data structures. For this reason, the first few invocations running interpreted code are typically slow. While code is interpreted, the runtime gathers important profiling statistics related to the application's execution (*e.g.,* how many times each function was executed). Then, after a certain number of invocations, these statistics are used to iteratively compile and optimize *hot* methods down to machine code. Even after a method has been optimized by the runtime, the runtime continues to gather code profiling statistics which can lead to advanced speculative optimizations tailored to specific application executions. Other similarly sophisticated runtimes, such as Python's PyPy [11] and JavaScript's V8 [34], use similar approaches based on code interpretation and iterative optimization that leverage application profiling.

To assess the importance of runtime techniques for iteratively optimizing code, we measured the latency of different function invocations for different serverless workloads from the SeBS [24] benchmark suite. In Figure 1, we show the function invocation latency for different request numbers for a Dynamic HTML generation workload when running on two different runtimes, PyPy (Python) and OpenJDK HotSpot (Java). In blue, we show where existing solutions trigger checkpointing. On the other hand, in orange, we show where Pronghorn targets checkpointing. We also summarized the latency speedup of function invocations for four benchmarks from [23] in Table 1. From this analysis, we draw three important observations that will guide Pronghorn's design.

Observation #1: runtime optimizations are highly effective in reducing the invocation latency of serverless functions, even across different compilers. For instance, for Dynamic HTML, the invocation latency of the first request (running unoptimized code) and a request running optimized code can differ by up to 33% (PyPy) and 76% (OpenJDK). We found that these significant speedups are also obtained for other serverless benchmarks, such as word counting, JSON parsing, and hashing. This confirms the benefits that modern runtimes bring to the performance of serverless workloads.

Observation #2: runtimes can take up to thousands of function invocations to reach a highly optimized version of the code. For instance, PyPy takes approximately 1000 requests to fully optimize the Dynamic HTML generation workload, while OpenJDK takes even longer, nearly 2500 requests (see Figure 1). This suggests that systems that checkpoint function instances close to instantiation [15, 27, 59, 61] will end up serving function invocations with suboptimal performance.

Observation #3: JIT compilers do not always generate code with better performance. JIT compilers optimize frequently invoked methods and often speculate to produce more efficient code, deoptimizing if any invariant is violated [19, 50, 53]. For example, the JIT compiler can produce code with the assumption that a particular variable will not be `null` and deoptimize if an invocation sees a `null` value. Method calls can also be speculatively de-virtualized if the callee is speculated to always be of the same type. If later a callee of a different type is used, the call site is deoptimized. Similar deoptimizations can happen when a method throws an exception for the first time. Upon a deoptimization, the runtime will gather additional profiling information before trying to re-optimize the function code. Subsequent rounds of optimization result in code being able to cover more code paths and therefore may result in lower performance [36].

To further complicate the problem, JIT compilation is not a deterministic process as it depends on a number of non-deterministic factors: which methods are already optimized (compilation is performed by background threads that contend for resources), code cache space availability, types already loaded, among others. Finally, JIT compilers have internal thresholds such as the size of a method, or the number of deoptimization rounds that, once hit, may prevent the method from ever be selected for optimization. As a result, JIT compilation is a complex process and even given the same set of profiles, the compilation of the same exact method is not guaranteed to generate the same optimized code.

Checkpointing cost. Another challenge is addressing the trade-off between the checkpointing benefits and cost. On one hand, more frequent checkpointing means we can collect a larger pool of function instances containing code optimized at different stages of a function's execution. This provides more chances of finding a function instance with higher performance. On the other hand, function checkpointing is an expensive task [73]. For instance, depending on the size of the function, checkpointing can take up to 100 ms and use tens of megabytes. This means that checkpoint-restore systems in the context of serverless need to be applied judiciously to not negate the benefits of running more optimized code.

Understanding code performance. Lastly, the effect of runtime optimizations on the function's end-to-end latency is not always immediate. This makes the task of measuring and understanding the progress being made by optimizations challenging. First, runtime optimizations do not always improve the performance of a function in a monotonic way, as illustrated in Table 1 across four different SeBS [24] benchmarks. For example, code de-optimizations may revert optimizations, making the execution temporarily slower. After collecting additional profiling information, the JIT compiler will re-optimize the code taking advantage of the additional profiling information. Second, runtime optimizations themselves can be expensive and interfere with the execution of the function. This means that simple checkpointing strategies that greedily checkpoint as long as invocation latency is decreasing will miss high-performance function snapshots.

3 Pronghorn

We present Pronghorn, a snapshot orchestration system for serverless platforms. In this section, we start by outlining the system's goals, model, and assumptions. Then, we discuss the general workflow of the system. Finally, we explain in detail Pronghorn's request-centric checkpoint orchestration policy and how it fulfills the proposed objectives.

3.1 System model and assumptions

Pronghorn's goal is to accelerate the average execution latency of serverless functions by continuously monitoring their performance, checkpointing function instances at specific moments, and launching functions from carefully selected snapshots.

To achieve generality and ease of deployment, Pronghorn leverages a set of black-box capabilities commonly found in today's serverless platforms:

- A transparent mechanism to measure the latency of each function request as an approximate proxy for the performance of a particular serverless function snapshot;
- One or more high-level language runtimes (*e.g.,* OpenJDK HotSpot JVM) which are transparently invoked within workers by the serverless platform to run serverless functions;
- Language runtime JIT compilers that are automatically invoked during a function's execution to perform optimizations on the serverless function's code.

Pronghorn's design relies on the following components (see Figure 2):

- A per-worker Orchestrator, responsible for enforcing a snapshot management policy that decides when to snapshot, which snapshot to restore from, and which snapshot to evict;
- A Checkpoint Engine responsible for checkpointing and loading snapshots;
- A global Object Store and Database to store function snapshots and execution metadata used to manage the snapshots, respectively.

3.2 System Workflow

Pronghorn accelerates the execution of serverless functions by continuously but selectively checkpointing JIT-optimized functions and by intelligently deciding which function snapshot to use for a new request. We now explain how Pronghorn influences a function's execution to achieve this goal (see Figure 2).

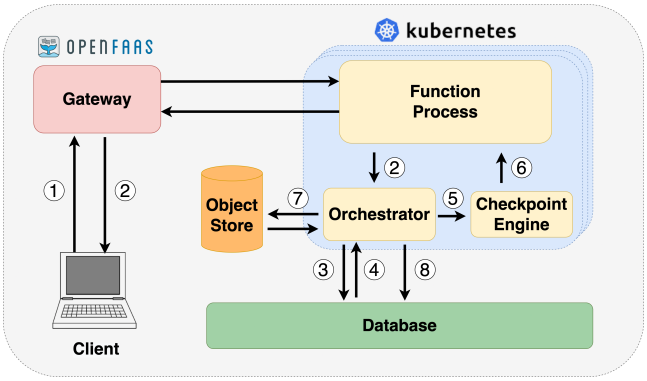

Figure 2. Pronghorn's design and execution steps (1-8) during the invocation of a serverless function.

To invoke a function, a client issues a request to the serverless platform through the platform's Gateway (①). Then, the serverless platform launches a container for its execution. When the worker starts, the Orchestrator is automatically invoked to determine what is the best snapshot to restore from for this particular function. Since the Orchestrator is local to the worker, it makes this decision based on global information about snapshots and their performances from the Pronghorn database. We explain how the Pronghorn policy leverages this information to make decisions in more detail in the next section.

If the policy decides to use an existing snapshot, the Orchestrator calls the Checkpoint Engine to start the invocation from that snapshot. To do this, the Orchestrator reads the snapshot from the Pronghorn's Object Store to the worker's local memory and restores the function process from the snapshot using the Checkpoint Engine. If the Pronghorn policy does not recommend using a snapshot — *e.g.,* the very first time the function is invoked — the Orchestrator performs a cold start of the workload process instead.

When the function's request terminates, the response of this execution is returned to the client through the Gateway (②). Then, the end-to-end latency is passed to the Orchestrator's policy, which uses this performance value to update the Pronghorn's Database (③). Since this function might have been executed in other workers concurrently with this invocation, the Orchestrator queries the Database (④) to update its own information about available snapshots. This information will be used in the next function invocation in this worker. If the orchestration policy determines a checkpoint is necessary, the Orchestrator directs the Checkpoint Engine to checkpoint the function's process (⑤). When invoked, the Checkpoint Engine creates a snapshot of the function process (⑥) and saves it to the local file system. The Orchestrator then uploads the compressed snapshot to the Pronghorn's Object Store (⑦) and records the location of the snapshot and relevant metadata in the Database (⑧).

3.3 Orchestration Policy Design

We now outline the design principles used in Pronghorn's orchestration policy.

Exploration-exploitation tradeoff. Part of the success of Pronghorn hinges on being able to find the function snapshots that provide the best performance. Unfortunately, it is not obvious from a small number of function invocations which snapshots can, over many invocations, perform the best. To make matters worse, it is hard to predict when the JIT compiler is ready to perform code optimizations. In fact, it can take hundreds of invocations before a function is optimized. For these reasons, Pronghorn needs to carefully balance the *exploration* of promising snapshots with the *exploitation* of the best-known ones.

Fixed-size snapshot pools. Generally, the more snapshots Pronghorn can store in the snapshot pool, the higher chances it has of finding high-performance snapshots. On the other hand, Pronghorn needs to keep overall costs (memory, storage, network) under control so they do not negate the performance benefits. For this reason, Pronghorn allows cloud providers to configure how much storage they want to allocate for snapshots (the snapshot pool size), and the system automatically takes advantage of that space to optimize the snapshot exploration policy.

Accurate performance estimation. In order for Pronghorn to decide which snapshots to explore, exploit, or evict from the pool, it needs to be able to accurately estimate the expected performance of a given snapshot. This task is complicated by the fact that the latency of a function invocation can vary significantly, even when using the same snapshot, due to differences in hardware, resource contention, or interference. Such differences are common in distributed environments. Furthermore, the performance of a particular snapshot can also vary significantly over time due to JIT compilation, which typically improves performance, but can also degrade it (as discussed in §2). Pronghorn tackles these issues in two ways. First, to account for performance variations due to changes in the environment, it aggregates performance measurements from different invocations of the same snapshot. Second, to tackle the unpredictability in performance from JIT compilations performed over many invocations, it evaluates the performance of each function for its total expected lifetime. To this end, Pronghorn monitors the performance across all invocations starting from a particular snapshot to build a complete picture of its performance.

Continuous learning. In practice, function performance often varies over longer periods of time due to changes in aspects as diverse as the workload inputs, the serverless environment (*e.g.,* network traffic, runtime versions), or even the configuration of the serverless functions (*e.g.,* resources

Symbol	Description
Precomputed:	
β	Average number of requests handled by a worker before eviction.
Overhead bounding:	
C	Maximum capacity of the snapshot pool.
W	Largest request number at which checkpointing is permitted.
Learning:	
α	Proportion for the exponentially weighted moving average update.
p	The top p% of snapshots are retained when maximum pool capacity is reached.
γ	γ% of randomly chosen snapshots are also retained when capacity is reached.

Table 2. Notation for Pronghorn's request-centric orchestration policy. *Precomputed* parameters are computed by the cloud provider and provided to the algorithm. *Overhead bounding* parameters are controlled by cloud providers based on their desired performance trade-offs. *Learning* parameters modify the speed and mode of convergence.

allocated to the functions). All of these can significantly affect the performance of the snapshots used by Pronghorn, and thus its end-to-end benefits. To adapt to these changes, Pronghorn continuously profiles the performance of snapshots and prioritizes recent information when making scheduling decisions.

3.4 Request-Centric Orchestration Policy

Pronghorn's orchestration policy (termed the "request-centric orchestration policy") is responsible for coordinating checkpoint and restore operations across all workers allocated to an application. It makes four key decisions: (1) when to checkpoint; (2) from which snapshot to restore a new container; (3) how many and which snapshots to store; and (4) how to update the orchestrator's state during execution. In the rest of the section we introduce the necessary notation, provide a pictorial overview of the algorithm, and walk through the algorithm step-by-step, explaining how it achieves our design principles.

Overview. Pronghorn's orchestration policy (illustrated in Figure 3, with accompanying notation in Table 2 and optional pseudocode in the Appendix) provides orchestration decisions at four specific moments in a function lifetime: (1) worker initialization, (2) worker startup, (3) every function request, and (4) snapshot pool capacity reached.

Every orchestration policy has two fundamental primitives: when to checkpoint an existing worker, and which snapshot to use to restore a new worker. Both of these primitives are driven by latency measurements — we prefer to restore a new worker from a snapshot that is taken at a request range with low latency rather than high latency, just as we prefer to take a snapshot at a request range with low

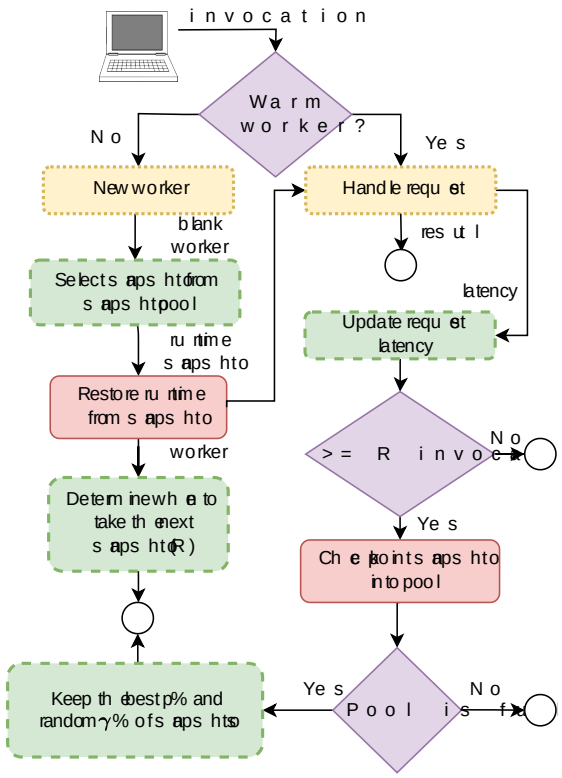

Figure 3. Pronghorn's request-centric policy flowchart overview. Green/dashed components represent core logic units of our policy; Red/solid components represent checkpoint and restore operations; Yellow/dotted components represent worker operations.

latency for future use. However, we need to gain latency measurements across a fixed request range in order to determine what constitutes a low latency. The fixed request range $[0, W]$ is our *search space* for this policy, with W defined in Table 2.

Knowledge updates. To persist knowledge, we first zero-initialize a weight vector of length W, which we use to store latencies for each request number. With every single request to an alive worker, we update the latency measurement for that request number in the weight vector (see Figure 3). This process enables us to learn latencies over multiple worker lifetimes. A crucial aspect of this policy is that rather than replacing the latency every single time a new measurement is given, we instead build up an exponentially-weighted moving average (EWMA) of request latencies in the request weight vector by taking a weighted average of the previous value and the new value (*i.e.*, $v_{\text{new}} = (1-\alpha)\, v_{\text{old}} + \alpha v_{\text{new}}$, with α defined in Table 2). By employing an EWMA, more recent samples in the time series are weighted higher while still retaining earlier knowledge. Through continuous improvement of latency estimates and prioritization of recent

samples, we expect our performance to be resistant to application input variance.

Snapshot restoration. With this weight vector formulation, deciding which snapshot to restore from may appear straightforward: pick the snapshot that, when restored from, has the lowest latency over an average lifetime (the *lifetime latency*, computed as an interval sum of the weight vector). However, multiple promising regions of the search space can exist simultaneously, and we still want to be able to traverse local optima, so we instead draw randomly from the snapshot pool, with each snapshot weight inversely proportional to lifetime latency. This process occurs in the first green box in the left path of Figure 3. We first invert the snapshot weights, and then convert the inverse weights into a probability distribution by applying the **softmax** [33] function.[2] The **softmax** function outputs a probability distribution over an input vector such that element values close to the maximum of the vector receive most of the entire weight, while the remaining elements receive the remainder. As a result, the snapshots with the lowest lifetime latency will be restored from most often, but even snapshots that have high lifetime latencies will still be restored from, albeit less often. This ensures that the policy will eventually explore even high-latency regions.

Checkpoint orchestration. We must also implement the second primitive: when to checkpoint an existing worker. We should checkpoint at lower-latency request numbers, but we also need to explore the request range. To achieve both goals, our policy will checkpoint a live worker at a certain request number over its lifetime. That request number is drawn from a probability distribution over the interval for which the worker is expected to be alive, with weight inversely proportional to its latency in the weight vector. This logic executes in the second green box on the left path in Figure 3. Since unexplored request numbers (for which we have no latency measurements) are zero by default, this request weighting scheme puts enormous weight on checkpointing at unexplored requests. Once no unexplored requests exist in the range $[0, W]$, it will preferentially checkpoint at low-latency request numbers. As a result, our policy addresses both of the goals outlined above.

Snapshot pool management. We implement an "exploration-exploitation" tradeoff by fixing a maximum capacity for our snapshot pool, and whenever that capacity is reached, evicting the worst-performing snapshots while also keeping a random subset. This logic occurs starting in the purple diamond at the bottom-right of Figure 3. Performance for a checkpoint is measured by summing learned request latencies over a worker lifetime (*i.e.,* β requests) when resuming from that checkpoint. Through repeated pruning and re-seeding of the snapshot pool, we expect the policy to converge on a desirable mixture of snapshots in the pool. A random subset is not deleted in order to enable more difficult optimizations that may necessitate crossing local optimization minima ("hill-climbing"). Similar bounded random exploration has proved successful in the realm of machine learning and reinforcement learning [38].

[2]For a vector $\mathbf{v}$, it is defined as $\mathbf{s} = \frac{1}{\sum e_i}\mathbf{e}$ where vector $\mathbf{e}$ is the result of applying the exp function element-wise to $\mathbf{v}$.

4 Implementation

We implement Pronghorn by building on OpenFaaS [10], a popular open-source serverless platform, allowing us to create an end-to-end serverless framework. We explain how we implement the components outlined in Figure 2 as follows (see §3.2 for an explanation of how the various components interact).

We implemented the Orchestrator in approximately 800 lines of Python as a Flask [4] application with an event-driven architecture. The Orchestrator manages snapshots in the Object Store (implemented with MinIO [8]) and stores policy weights in the Database. The Orchestrator also forks and executes the function process and determines when to checkpoint and from which snapshot to restore based on its policy. When checkpointing or restoring, the Orchestrator utilizes CRIU [25] as the Checkpoint Engine to create the snapshot of the function process, upload it to MinIO, and update the policy weights in the database. Furthermore, the Orchestrator can terminate the function process at predefined request intervals (simulating cloud providers evicting workers) to analyze Pronghorn's performance across various traffic patterns (see §5.2).

We employed CRIU [25] as a stand-in for any Checkpoint Engine due to its maturity and widespread usage. However, Pronghorn is agnostic to the choice of Checkpoint Engine. This enables us to compare orchestration strategies across any checkpoint-restore system with our own strategy. A corollary of this approach is that the benefits of our orchestration strategies can accrue to serverless systems that use different checkpoint-restore implementations (*i.e.,* those which have custom checkpointing or restoring solutions).

To provide flexibility and support various orchestration policies, we designed the Orchestrator to execute policies through a minimal abstract interface, enabling easy implementation of a range of policies. At its core, the policy must implement interface functions that dictate which snapshot to use when starting a new worker and when to checkpoint a running worker.

The Database is a lightweight implementation of a general-purpose key-value store. It is a Flask [4] application written in approximately 100 lines of Python. The Database interface provides a persistent store for storing orchestration policy weights and learned data, exposing only strongly-consistent atomic read and write operations. Since the Database only

Benchmark	Description
Java:	
`HTMLRendering`	HTML template rendering with random numbers
`MatrixMult`	Square matrices multiplication with random sizes
`Hash`	Checksum of a large random bytes array
`WordCount`	Word count for random-length excerpts
Python:	
`BFS`	Breadth-first search on random graph
`DFS`	Depth-first search on random graph
`MST`	Minimum spanning tree (MST) of random graph
`DynamicHTML`	HTML generation with randomized content
`PageRank`	PageRank [51] on random graph
`Uploader`	Upload file from provided URL to cloud storage
`Thumbnailer`	Generate a thumbnail of an image
`Video`	Add a watermark and generate gif of a video file
`Compression`	Create a .zip file for a group of files in storage

Table 3. Evaluation benchmarks.

needs to implement a minimal interface, production key-value stores (*e.g.,* Redis [55], Dynamo [26]) can be readily substituted.

In addition to these components, we also implemented infrastructure to enable benchmarking Pronghorn. Our benchmarking infrastructure for PyPy benchmarks is approximately 800 lines of Python code, and the infrastructure for JVM runtime-based benchmarks accounts for approximately 450 lines of Java code.

5 Evaluation

We evaluate Pronghorn by presenting an analysis of our proposed orchestration policy compared to a baseline scheduling strategy and the state-of-the-art scheduling strategy on common serverless workloads. We quantify the benefits of these strategies, reporting the distribution of end-to-end request latencies across multiple container eviction rates and language runtimes. To demonstrate resilience to perturbations in inputs, we artificially introduce significant variance in benchmark inputs to better model the range of inputs a live serverless application could receive. Finally, we evaluate the costs and overheads associated with the deployment of Pronghorn.

5.1 Methodology

Benchmarks. Our evaluation encompasses a set of thirteen benchmarks adapted from representative benchmarking suites such as ServerlessBench [71], FaaSDom [46], SeBS [24], and previous papers [23]. These benchmarks were implemented using Java 17 and Python 3, as outlined in Table 3. The benchmarks were executed on two popular runtimes: the Java Virtual Machine (JVM) and the PyPy Just-in-Time (JIT) compiler. The selection of the JVM as one of our target runtimes was driven by its widespread adoption as a prevalent target platform for a multitude of high-level languages. Its extensive usage in industry and its robust optimization pipeline make it an ideal choice for our evaluation. Additionally, PyPy was chosen as the JIT compiler for Python, considering its prominent position as one of the most widely used high-level languages in scientific computing and scripting. To ensure accuracy and to encompass a diverse range of workloads, our evaluation encompasses both compute and I/O-bound tasks. By including a variety of benchmarks that represent different types of workloads, we aim to provide a comprehensive assessment of our system's performance.

In order to ensure the robustness of our system to perturbations in benchmark inputs, we made slight modifications to each benchmark, adding optional zero-mean Gaussian noise in the inputs of up to an order of magnitude in the input sizes. This adjustment is particularly significant for graph-based benchmarks, as the execution latency directly scales with the size of the random graph, which is the primary input to the benchmark. By introducing high variance in inputs, we can determine whether our tested policies are resistant to noise in inputs and, consequently, in request latencies.

Measurements. To provide a holistic view of the system performance, all latency measurements are end-to-end and are measured from the perspective of the client. We measured the end-to-end latency for 500 invocations of each workload for three orchestration policies. In our evaluation, we present the Cumulative Distribution Functions (CDFs) of these latencies to provide a comprehensive analysis of latency distribution across the entire set of 500 requests. By examining the CDFs, we aim to provide a holistic understanding of the system performance. To ensure a comprehensive evaluation, all orchestration strategies were subjected to benchmarking with eviction rates of 20, 4, and 1 request per worker. These rates roughly correspond to a request issued every 1 minute, 5 minutes, and 1 hour, respectively, assuming an average worker lifetime of approximately 20 minutes based on real-world data [58].

Measurements were conducted on a three-node VM cluster running Ubuntu 22.04 LTS, equipped with 2 vCPUs, 8 GiB memory, and 50 GiB of disk space. The VM cluster was hosted on a bare-metal machine running Ubuntu 20.04.4 LTS, with 61 GiB of RAM and a quad-core 4.0 GHz Intel i7-6700K CPU.[3] We employed k3s [6] (a lightweight Kubernetes [7] distribution) for cluster management and Docker [3] for containerization. The execution environment consistently utilized OpenJDK 17 and CRIU 3.15. We adopted a multi-VM configuration to better resemble real-life serverless systems, with one VM serving as the control plane and the other two operating as workers. We note that Pronghorn is not limited

[3]While more specialized hardware exists for large-scale virtualization, we opted for commodity server-grade hardware sufficient to run a mid-scale OpenFaaS setup, which we believe is reasonably representative.

to a three-node system — it generalizes to a distributed system with several nodes in the control plane managing a fleet of possibly hundreds of workers. In our setup, the Docker server runs on the host machine, and the user launches and invokes functions from there. When functions are launched, they are executed within the designated workers (in our case, Docker containers).

Orchestration policies. We evaluate three orchestration policies:

1. **Cold-start:** Starting the workload anew each time a worker is initialized (no checkpoint-restore);
2. **Checkpoint after 1st** (*state-of-the-art*): Checkpointing immediately after the first request is complete, and resuming from that snapshot hereafter. State-of-the-art systems including Catalyzer [27], Fireworks [59], Prebaking [61], Groundhog [15], and Amazon Lambda's SnapStart [16] snapshot after the first request or after initialization but before the first invocation. We note that checkpointing after initialization and before the first invocation results in inferior performance as runtimes lazily initialize many internal data structures inside the interpreter and JIT compiler, meaning that much initialization will happen when the first request is executed;
3. **Request-Centric**: Request-centric checkpointing (as described in §3.4) with $p = 40\%$, $\gamma = 10\%$, and snapshot pool capacity $C = 12$. We set W to 100 for PyPy benchmarks and 200 for JVM benchmarks.

We chose values of p and γ that seemed reasonable to prevent overfitting. A snapshot pool capacity of $C = 12$ comes out to around 600 MB using average checkpoint sizes from Table 4, which is under 1 GB, the lowest unit AWS S3 [1] uses.

Stopping condition. Since each benchmark-strategy combination was evaluated for 500 invocations, bounding the search space to 100 requests (*i.e.,* setting $W = 100$) assured that the first 100 requests would be fully "explored". For the JVM, we used $W = 200$ since the JVM generally takes twice as long as PyPy to arrive at an optima (as seen in Figure 1). While we did not stop checkpointing workers after some fixed number of requests for the purposes of our evaluation, the cloud provider can always choose to stop checkpointing and use the best snapshot available in the pool thereafter. For example, if the cloud provider observes Pronghorn is not improving performance throughout several worker lifetimes, then it can halt further exploration for that application.

5.2 End-to-End Evaluation

We analyze the immediate impact of our orchestration strategy and then examine the impact of different traffic patterns. As explained in §5.1, we demonstrate the resilience of our orchestration strategy to perturbations in the input by running all benchmarks with significant artificial variance in input sizes, thereby inducing variance in request latencies.[4]

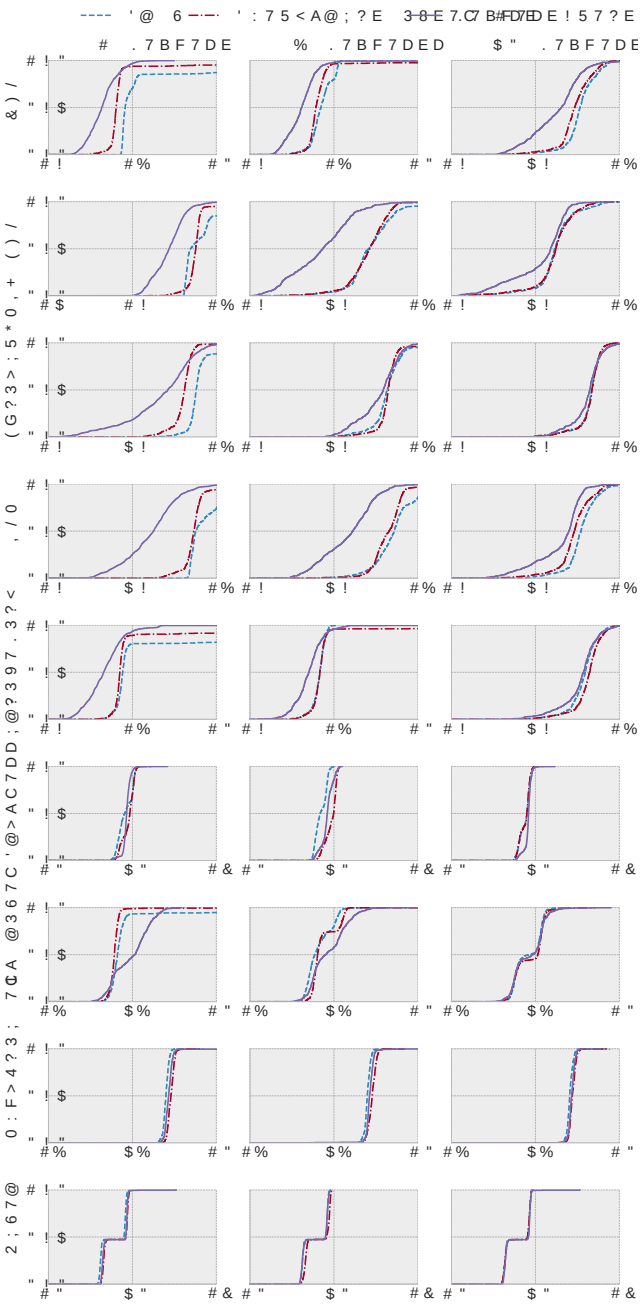

Figure 4. The Cumulative Distribution Function (CDF) of end-to-end request latency in microseconds of Python benchmarks (rows) across the evaluated orchestration strategies and three different container eviction rates (columns).

[4]Evidence of the large variance can be seen in the provided latency CDFs for compute-bound benchmarks, where a typical request interquartile range spans over an order of magnitude.

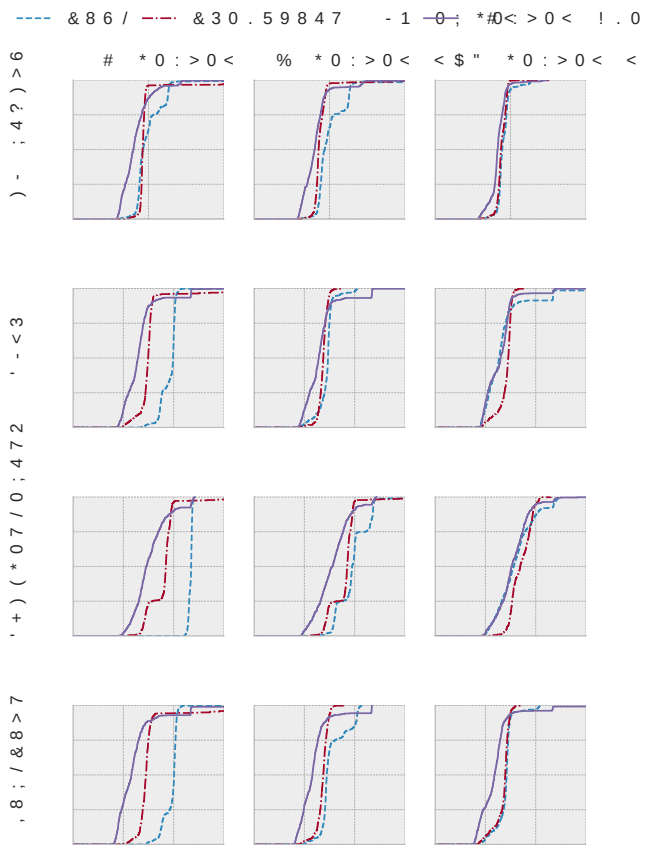

Figure 5. The Cumulative Distribution Function (CDF) of end-to-end request latencies in microseconds for all Java benchmarks (rows) across the evaluated orchestration strategies and three different container eviction rates (columns).

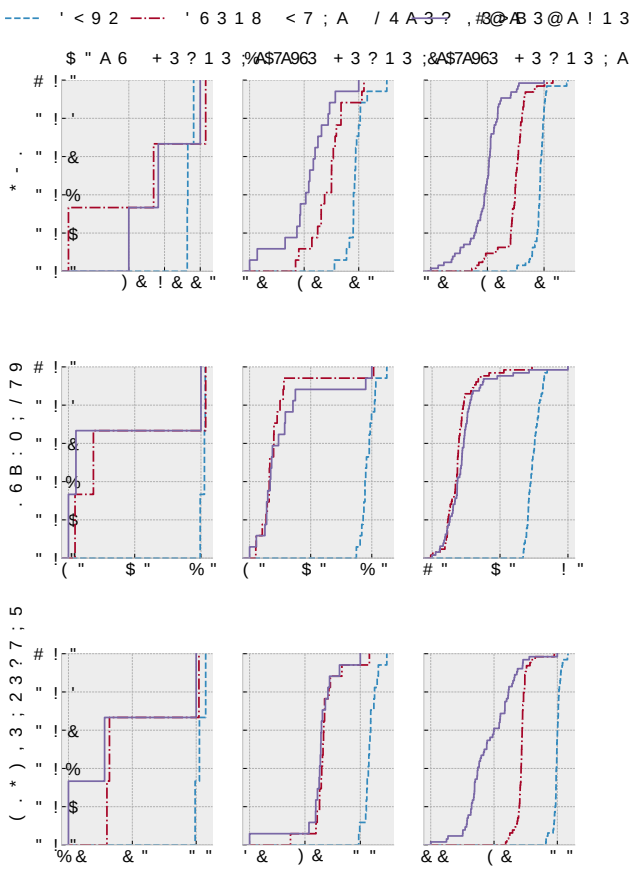

Figure 6. The Cumulative Distribution Function (CDF) of end-to-end request latencies in microseconds for two compute-bound workload and an I/O-bound workload for varying percentiles of function popularity sampled from fifteen minutes of the Azure production traces.

Orchestration strategy. Figure 4 and Figure 5 report the distribution of request latencies of thirteen different workloads across two popular language runtimes (PyPy and OpenJDK HotSpot JVM, respectively) across three different strategies. In this paragraph, we examine the *first* column (where the function workers are evicted after every request) and compare Pronghorn's performance to the state-of-the-art. Given that only a quarter of typical serverless functions receive more than one invocation in 10 minutes [58] (the typical worker eviction timeout), this frequency of evictions applies to nearly three-quarters of serverless functions. For six of the thirteen benchmarks (`BFS`, `DFS`, `DynamicHTML`, `MST`, `PageRank`, and `WordCount`), our policy provides a significant improvement at every percentile over the state-of-the-art (note that the x-axis is log-scale), with a median improvement ranging from 20.5% to 58.9%. For three of the benchmarks (`MatrixMult`, `Hash`, and `HTMLRendering`), our policy provides a clear benefit up until the 90th percentile of request latencies, with a median performance improvement of 24.8%, 36.8%, and 58.9% respectively. The remaining four benchmarks are IO-bound rather than compute-bound: for three of them (`Compression`, `Thumbnailer`, and `Video`) our policy provides on-par performance (within 5% of state-of-the-art), while only one (`Uploader`) shows worse performance. The benchmark that shows worse performance (`Uploader`) is entirely IO and network bound since the actual computation is performed by calling out to a native C library, so the benefit from runtime optimizations and Pronghorn is marginal. Overall, for a typical serverless request density, Pronghorn improves upon median performance for cold-start and the state-of-the-art policy for 9 of 13 benchmarks, with a geometric mean of improvement (based on percentage improvement in median) of 37.2% over the state-of-the-art, worsens performance for 1 of 13, and provides on-par performance for 3 of 13 benchmarks.

Request rates. Comparing performance across the columns of Figure 4 and Figure 5, we can analyze the robustness of the performance improvements produced by Pronghorn's request-centric strategy for workloads with different request traffic patterns. Specifically, the tested eviction rates (in the figure columns) of every 1, 4, or 20 requests correspond to a request issued every 1 hour, 5 minutes, or 1 minute, respectively. We observe that across the 39 settings, Pronghorn almost always performs better, or at least as well as the state-of-the-art. In 28 of 39 experiments, Pronghorn provides better median performance (a total geometric mean of improvement of 22.5%); in 9 of 39, Pronghorn provides on-par median performance (within 5% of state-of-the-art); and in only 2 of 39 experiments does Pronghorn clearly lower median performance (`Uploader` for eviction rate of 1 and 4) for the reasons mentioned in the previous paragraph. Of the benchmarks where Pronghorn provides better median performance, the geometric mean of improvement was 37.2% for eviction rate 1, 22.5% for eviction rate 4, and 13.5% for eviction rate 20. This shows that the sooner function workers are evicted, the larger the performance improvements yielded by Pronghorn. Moreover, this confirms our motivation for the system — that valuable runtime optimizations are lost when function workers are frequently evicted, and that preserving those optimizations leads to significant performance improvements. The data show that Pronghorn is resilient to differing request traffic patterns, which is crucial for Pronghorn to be deployed in a production setting.

Trace analysis. We additionally evaluate the performance of different snapshot orchestration strategies using a set of three 15-minute function traces extracted from Azure production traces [58]. Each trace contains all invocations of a single function for a fixed-size time window of fifteen minutes. Functions were selected uniformly at random after filtering by their popularity percentile, measured in number of invocations per day.

Results in Figure 6 demonstrate that Pronghorn still outperforms other orchestration strategies as it (1) offers superior performance in 6 out of 9 scenarios, (2) is on-par with other strategies in 2 scenarios, and (3) only degrades performance in 1 pathological scenario. This pathological scenario (on the `MST` benchmark for 50th percentile of popularity) occurs due to an extremely low number of invocations (3 requests) in a fifteen-minute window. Given more invocations over a longer period, Pronghorn would converge to a better snapshot.

5.3 Cost Analysis

We analyze the costs that can be expected from Pronghorn in a production deployment and explain how to bound these overheads. To that end, we measure the overheads introduced by Pronghorn compared to the state-of-the-art and discuss practical strategies to both bound and mitigate them.

Benchmark	Req. #	Checkpoint (ms)	Restore (ms)	Snapshot (MB)
Java:				
`HTMLRendering`	215	70.7 ± 25	50.4 ± 5.8	10.5
`MatrixMult`	203	66.1 ± 11	51.5 ± 3.9	10.6
`Hash`	212	60.6 ± 13	52.5 ± 3.8	10.6
`WordCount`	218	67.9 ± 18	55.2 ± 4.0	13.3
Python:				
`BFS`	100	85.6 ± 21	73.8 ± 9.5	55.5
`DFS`	114	85.7 ± 21	70.8 ± 13	55.8
`MST`	135	79.6 ± 23	77.1 ± 2.1	56.1
`DynamicHTML`	287	74.4 ± 22	75.3 ± 6.5	54.1
`PageRank`	100	74.4 ± 16	80.5 ± 7.2	64.0
`Uploader`	154	100.2 ± 13	30.2 ± 2.4	61.2
`Thumbnailer`	100	100.7 ± 14	67.0 ± 6.3	62.0
`Video`	193	91.1 ± 12	40.4 ± 2.4	60.1
`Compression`	136	105.0 ± 8.0	39.1 ± 1.3	61.0

Table 4. For each benchmark, the requests taken by Pronghorn to find the optimal snapshot; snapshot sizes; and checkpoint/restore times. Request number is computed by sliding a window of size 20 across the recorded latencies to find the interval whose median is within 2% of the final value. The reported request number is the mean of convergence request numbers across all tested combinations of input size variances and eviction rates for each workload. Checkpoint and restore timings and sizes came from repeatedly checkpointing and restoring each benchmark 10 times after startup and taking the mean across those 10 runs on the same machine used for evaluation.

Checkpoint-restore overhead. Pronghorn's request-centric strategy first undergoes an exploration period, configured by the hyperparameter W, before honing in on an optimum. Therefore, the number of checkpointing primitives is bounded: Pronghorn only incurs the fixed costs for checkpointing until it stops exploring.[5] Once an optimal snapshot is found for an application, it can be deployed with that snapshot indefinitely. Therefore, in serverless functions with high request rates, the recurring speed-up due to Pronghorn will rapidly accumulate, while checkpointing overhead will cease entirely once exploration stops.

The checkpointing primitive of Pronghorn causes a brief worker downtime on the order of 60 − 105 ms when using CRIU 3.15 without any additional optimizations, as seen in Table 4. For the same choice of Checkpoint Engine, the restoration overhead is the same as that experienced by state-of-the-art systems [27, 61] that rely on a similar primitive to increase an application's performance. Every system that makes use of checkpoint-restore will incur the restore cost, so Pronghorn does not increase restore costs relative to the state-of-the-art — Pronghorn only increases the number of checkpoints taken. Moreover, checkpointing overhead is

[5]The cloud provider can choose when to stop checkpointing — the best snapshot seen will always be in the snapshot pool.

Benchmark	Max Storage Used (MB)	Max Network Used (MB)	Baseline Storage Used (MB)	Baseline Network Used (MB)
Java:				
HTMLRendering	126	2625	10	1312
MatrixMult	127	2650	10	1325
Hash	127	2650	10	1325
WordCount	159	3325	13	1662
Python:				
BFS	666	13875	55	6937
DFS	669	13950	55	6975
MST	673	14025	56	7012
DynamicHTML	649	13525	54	6762
PageRank	768	16000	64	8000
Uploader	734	15300	61	7650
Thumbnailer	744	15500	62	7750
Video	721	15025	60	7512
Compression	732	15250	61	7625

Table 5. For each benchmark, the maximum storage used by Pronghorn's orchestration strategy, the maximum cumulative network bandwidth used to transfer snapshots, and the baseline values for state-of-the-art. Baseline storage used is the average snapshot size for the benchmark; maximum storage use is computed as the product of the former and the size of the snapshots pool, C. The maximum network use is computed as double the product of the total number of container lifetimes and the average snapshot size for the benchmark, while baseline is half that. The doubling is due to the network transfer cost of uploading a snapshot.

not directly experienced by the user, *i.e.,* workers are only checkpointed after a request has been completed. Analysis of production traces [58] reveals that most deployed functions handle few requests over long periods, while the most frequently-invoked account for the majority of compute time. If the application is infrequently invoked, then this brief worker downtime will not impact user latency, as invocations during that downtime are unlikely. On the other side of the spectrum, serverless functions in heavily distributed contexts with high request density are typically auto-scaled by the serverless platforms. As a consequence, their respective load balancers can direct requests away from workers which are being checkpointed, thus shielding the user from that latency in both cases.

Storage and network overheads. Pronghorn will generally improve performance and use less compute, at the expense of the additional storage and network bandwidth used. The benefits will accrue to both the user and the cloud provider, while the storage and network costs will be primarily borne by the cloud provider.[6] Table 5 reports the maximum storage and network bandwidth used by Pronghorn's

[6]Depending on their pricing model, cloud providers may choose to pass on some portion of the cost to users.

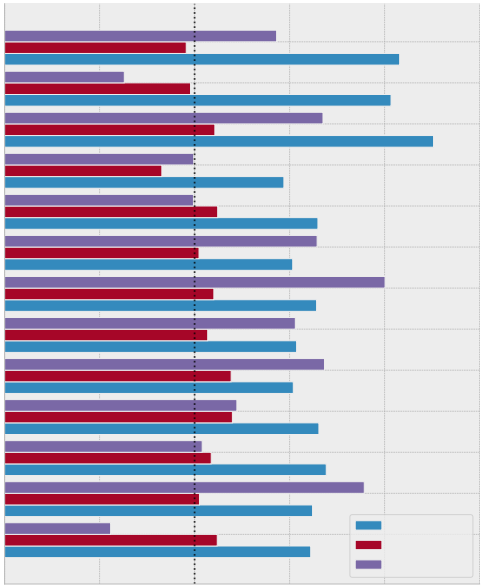

Figure 7. Per-operation overheads of Pronghorn for the Request-Centric strategy versus the baseline strategy (Checkpoint after 1st), across three components of orchestration: per worker startup, per request, and per checkpoint. Each benchmark is separately normalized, first against the baseline and second against the number of relevant operations. To normalize against the number of operations, the total worker startup overhead is divided by number of worker lifetimes; the total checkpoint overhead is divided by number of checkpoints taken; and the total request overhead is divided by number of requests served. Dashed line marks equality with baseline.

orchestration policy and those of the state-of-the-art policy. Compared to the state-of-the-art, Pronghorn will use twice the network bandwidth during the exploration period (per container lifetime, one restore, and one checkpoint operation, each requiring a snapshot transfer over the network). After the exploration period, Pronghorn will incur the same network bandwidth cost as the state-of-the-art (one restore operation per container lifetime). In terms of storage, Pronghorn will use up to the size of the snapshot pool (*i.e., C* snapshots) and therefore up to C times the storage of the state-of-the-art policy. Network and disk operations associated with snapshot creation and transfer do not impact user-perceived latency, as they occur asynchronously with respect to the request flow. However, these operations can increase the load on the whole system.

Bounding system costs. Table 4 also illustrates the number of requests the request-centric policy takes to converge to an optimal snapshot in our main evaluation experiments. For both PyPy and JVM, we observe that the request-centric

policy converges in less than $W + 100$ requests for every single benchmark (300 requests for JVM benchmarks and 200 requests for every PyPy benchmark). Consequently, the cloud provider can stop further checkpointing after $W + 100$ invocations of the application, avoiding any additional network bandwidth costs or checkpointing downtime compared to the state-of-the-art policy.[7] Of course, cloud providers may wish to navigate the Pareto frontier between performance and exploration cost differently, and so Pronghorn enables them to choose their desired trade-off by simply ending the exploration period whenever they choose. The cloud provider can also directly lower the storage overhead used by simply reducing the size of the snapshot pool (*e.g.,* setting $C = 2$ instead of $C = 12$).

Checkpointing overheads can be further mitigated when serverless applications are run in a distributed context (*e.g.,* for a frequently-invoked application running on numerous containers). Only a nonempty subset of containers running a given application need to be exploring in order to realize performance benefits — the remaining containers can simply restore from the best snapshots found so far. Exploration overheads can therefore be amortized over many containers, with the degree of amortization chosen by the cloud provider.

Orchestrator overheads. Figure 7 reports per-operation overheads associated with the main decision points in Pronghorn's orchestration algorithm (worker startup, request processing, and checkpointing), normalized against the Checkpoint after 1st strategy as the baseline. Overheads were tabulated for a full run with the same benchmarking suite and methodology as described in subsection 5.1. Importantly, while these overheads place load on the system, they all occur off the critical path of end-to-end request processing, and are thus not directly observed by the user. Container startup overhead is higher than the baseline (although not exceeding 2.5x, or 28ms), which is expected given that Pronghorn must decide from which snapshot to restore, a decision that is trivial for the baseline strategy. Per-request processing is generally on-par with the baseline, as Pronghorn only incurs some extra array read-write operations, whose computation time is outweighed by network latency. Finally, the per-checkpoint time is higher for 9 of 13 benchmarks (not exceeding 2x, or 34ms) as Pronghorn has to update the snapshot pool in the database. We note that the orchestrator implementation was unoptimized (*i.e.,* written in Python with many logging operations), and could be substantially improved.

[7] A corollary of this empirical bound is that serverless applications that receive very few requests (less than 150 requests, assuming a small $W < 50$) over the duration of their deployments are unlikely to benefit from Pronghorn.

6 Discussion

In this section, we discuss additional aspects that can affect Pronghorn's effectiveness, and sketch potential improvements for future work.

Lifetime estimation. We assume that cloud operators can provide a reasonable estimate of the lifetime of a function worker. Should this estimate be inaccurate, there are two possibilities: worker lifetime can be underestimated or overestimated. An underestimate means we believe the worker will die sooner than it actually does, and so we checkpoint sooner than is ideal, thereby taking longer to explore the request range. Conversely, an overestimate means we believe the worker will be evicted later than it actually does, so we may plan to checkpoint at a certain number of invocations that is never reached. While this situation is not ideal, cloud providers will likely run numerous function workers over the entire deployment of a serverless application, and most likely some of them will regularly reach the predicted lifetime.

Tuning Pronghorn. Pronghorn can be tuned by cloud operators based on various factors, including estimated container lifetime, average JIT compiler warmup duration, and snapshot budget (network, storage, etc.). Adjusting the request-centric strategy parameters (*e.g.,* β, W, and C respectively for the prior examples) is sufficient to tune Pronghorn. By doing so, Pronghorn can converge faster and even work with a smaller snapshot pool. However, this approach may require prior knowledge about serverless function code, invocation patterns, or JIT compiler internals.

Workload and input-awareness. For serverless applications with multiple traffic patterns (*workloads*), different orchestrators can be specialized towards specific patterns. By doing so, instances can specialize for certain workloads, and thereby achieve a closer "fit" to the data rather than forcing a single snapshot to handle all workloads a function is subject to. Moreover, distinct inputs to serverless applications can lead to divergent code paths and execution profiles. While Pronghorn should converge to a mixture of snapshots that perform the best on average, across the distribution of request types and code paths, using Pronghorn to specialize snapshots for specific code paths is an exciting future work direction. We envision a meta-optimization strategy that utilizes entirely different policies based on patterns in workloads and inputs to improve global response latency.

7 Related Work

Recent research efforts have yielded numerous systems that leverage checkpoint-restore mechanisms to improve serverless platforms. Silva et al. [61] employ a snapshot to restore the memory and state of a previously executed function runtime. Catalyzer [27] reuses user-level and system-level state by using a combination of a snapshot and a custom

OS primitive, reducing overall workload latency. Medes [57] deduplicates redundant memory footprints to develop sandbox state, which is faster to restore compared to a cold start. SEUSS [20] reuses state from unikernel and function-specific snapshots to expedite the deployment of serverless functions. These systems checkpoint the function runtime right after runtime initialization [15, 20, 27, 30, 67] or after the first function invocation [61], and, thus, miss the optima produced beyond the lifetime of a single container due to JIT runtime optimizations. Pronghorn can be deployed on top of these systems as a snapshot orchestration layer.

Fireworks [60], on the other hand, triggers early JIT compilation before the function code is even executed. This system takes advantage of Numba [42] and Node.js [9] capabilities to convert Python and JavaScript function code into machine code, respectively. By using Python and JavaScript JIT compilers as Ahead-of-Time compilers, code produced by premature JIT invocation misses important optimizations that rely on profiling dynamic code execution and which are known to be crucial for performance in high-level languages [28, 35, 63]. Instead of forcing JIT activity, Pronghorn does not interfere with the code compilation pipeline, allowing runtimes to fully optimize the code using dynamic profiling information.

Existing approaches that keep containers alive [14, 32, 44, 47, 49, 70] necessarily incur high costs to the cloud provider even if carefully optimized, since provisioning containers is expensive. Pronghorn provides high performance to the end-user while still retaining cloud providers' flexibility on when to evict containers. Similarly, Pronghorn does not rely on co-location of functions within containers or machines, which prior systems [29, 64] do to increase performance.

Pagurus [43] is a container management system that provides an orchestration layer on top of checkpoint-restore solutions like Catalyzer [27] in order to optimize resource utilization. Pagurus exploits inter-workload similarity by sharing warm containers across different applications. Therefore, Pagurus is orthogonal to Pronghorn's approach of optimizing the latency of one serverless workload at a time. In fact, Pronghorn can be modified to also resume from a warm container running a similar workload. However, to retain the performance benefits of intelligent orchestration, we would only do so when Pronghorn has no previous snapshots for a given application.

A recent work proposed a JIT server that shares the compilation effort across multiple runtimes [12]. However, this solution came at a high engineering effort, making it challenging to adopt in other language runtimes. Moreover, since profiling information is not shared in JITServer, a single runtime would need to receive a significant amount of requests in order to successfully profile and compile user code, which is unlikely in the face of frequent container evictions.

REAP [65] pre-fetches resident memory pages from disk when a serverless function is restored from a snapshot, thereby improving cold-start delays significantly. CheckSync [40] integrates with the function runtime to greatly decrease the size of function snapshots. LambdaLite [69] accelerates the application code loading phase in order to reduce the cold-start response latency of a function. CRaC [13] aims to make Java programs aware of checkpoint or restore operations to optimize snapshot sizes for Java programs. On-demand fork [73] provides a fast fork implementation by generalizing copy-on-write to page tables that can be used to accelerate process restoration. These optimizations can be applied in concert with Pronghorn due to it being agnostic to the underlying checkpoint engine and runtime. Azul ReadyNow [2] reads from JVM profiler logs in order to accelerate warm-up. By specializing for the JVM, ReadyNow achieves hot-start performance at a low cost, but unlike Pronghorn, is neither generalizable to other compiler versions nor runtimes.

To the best of our knowledge, Pronghorn is the first technique to orchestrate serverless checkpointing in order to automatically find the snapshot with the highest performance. Pronghorn is platform and runtime agnostic and can be easily integrated into production serverless platforms. Moreover, Pronghorn is orthogonal to many techniques proposed to reduce memory function footprint [18, 29, 39], reduce cold start latency [48, 56], enable networking [31, 66, 68], transactional serverless workloads [72], serverless workflows [45], and fast data exchange [41, 54, 62], among others.

8 Conclusion

Pronghorn enables serverless cloud providers to make full use of speculative JIT optimizations in modern runtimes. Pronghorn's novel approach builds up knowledge of the request latency patterns for a given serverless application, and leverages that knowledge to hone in on the best-performing function snapshot. Pronghorn can be readily integrated into existing serverless systems because it is agnostic to the underlying JIT runtime, platform, and checkpoint engine. Our experiments demonstrate that Pronghorn outperforms the baseline of no checkpointing and the state-of-the-art of checkpointing right after initialization, and maintains these performance gains in the face of high variance in request latencies and different traffic patterns. Finally, Pronghorn provides these benefits in request latencies indefinitely, while its costs are strictly bounded, and thus amortized over time.

9 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Antonio Barbalace, for their insightful feedback. This work was funded in part by the National Science Foundation (NSF) under grant CNS-2140305 and the Fundação para a Ciência e a Tecnologia under award UIDB/50021/2020.

References

[1] AWS S3. https://aws.amazon.com/s3/.

[2] Azul prime platform readynow. https://www.azul.com/wp-content/uploads/Prime-Data-Sheet-ReadyNow-Orchestrator.pdf.

[3] Docker. https://www.docker.com/.

[4] Flask. https://flask.palletsprojects.com/en/2.2.x/.

[5] JVM JIT-compiler overview. https://cr.openjdk.java.net/~vlivanov/talks/2015_JIT_Overview.pdf.

[6] K3S. https://k3s.io/.

[7] Kubernetes. https://kubernetes.io/.

[8] MinIO: High performance, kubernetes native object storage. https://min.io/.

[9] Node.js. https://nodejs.org/en/.

[10] OpenFaaS. https://www.openfaas.com/.

[11] PyPy. https://www.pypy.org/.

[12] JITServer: Disaggregated caching JIT compiler for the JVM in the cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[13] Openjdk crac project. https://wiki.openjdk.org/display/crac, Jul 2022.

[14] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[15] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 398–415, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Amazon. Improving startup performance with Lambda SnapStart. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html, 2023.

[17] Håkan Ardö, Carl Friedrich Bolz, and Maciej FijaBkowski. Loop-aware optimizations in pypy's tracing jit. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, page 63–72, New York, NY, USA, 2012. Association for Computing Machinery.

[18] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[20] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Proceedings of Workshop on Systems for ML at NeurIPS (MLSys'18)*, 2018.

[22] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ML workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'19, pages 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: Leveraging runtimes for the serverless era. HotOS '21, page 58–64, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing, 2020.

[25] CRIU. Checkpoint/restart in userspace (criu). https://criu.org/, 2021.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, oct 2007.

[27] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. Usetl: Unikernels for serverless extract transform and load why should you settle for less? In *APSys '19*, 2019.

[31] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[32] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Bolin Gao and Lacra Pavel. On the properties of the softmax function with application in game theory and reinforcement learning, 2017.

[34] Google. V8 JavaScript engine. https://v8.dev/, 2023.

[35] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 239–250, New York, NY, USA, 2012. Association for Computing Machinery.

[36] Tobias Hartmann. The Java HotSpot VM Under the Hood. https://cr.openjdk.org/~thartmann/talks/2017-Hotspot_Under_The_Hood.pdf, 2018.

[37] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. *SIGPLAN Not.*, 35(10):294–310, oct 2000.

[38] Sheldon Jacobson and Enver Yücesan. Analyzing the performance of generalized hill climbing algorithms. *J. Heuristics*, 10:387–405, 07 2004.

[39] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Nicolaas Kaashoek and Robert Morris. Checksync: Using runtime-integrated checkpoints to achieve high availability.

[41] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for

serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[42] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[43] Zijun Li, Quan Chen, and Minyi Guo. Pagurus: Eliminating cold startup in serverless computing with inter-action container sharing, 2021.

[44] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach, 2019.

[45] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[46] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pages 73–84, 2020.

[47] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[48] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[49] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[50] Scott Oaks. *Java Performance: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2014.

[51] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[52] Richard Plangger and Andreas Krall. Vectorization in pypy's tracing just-in-time compiler. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, page 67–76, New York, NY, USA, 2016. Association for Computing Machinery.

[53] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[55] Redis. Redis. https://redis.io/.

[56] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 753–767, New York, NY, USA, 2022. Association for Computing Machinery.

[57] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[59] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[60] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[61] 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, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[62] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[63] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, page 180–195, New York, NY, USA, 2001. Association for Computing Machinery.

[64] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 19–24, New York, NY, USA, 2019. Association for Computing Machinery.

[65] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[67] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Mike Wawrzoniak, Ingo Müller, Rodrigo Bruno, and Gustavo Alonso. Boxer: Data analytics on network-enabled serverless platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR 2021)*, 2021.

[69] Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, Xin Jin, and Xuanzhe Liu. Lambdalite: Application-level optimization for cold start latency in serverless computing, 2022.

[70] Bowen Yan, Heran Gao, Heng Wu, Wenbo Zhang, Lei Hua, and Tao Huang. Hermes: Efficient cache management for container-based serverless computing. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, Internetware '20, page 136–145, New York, NY, USA,

2020. Association for Computing Machinery.

[71] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.

[72] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020.

[73] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 540–555, New York, NY, USA, 2021. Association for Computing Machinery.

A Appendix

Algorithm 1 Request-centric orchestration policy algorithm

Initialize snapshot pool **P** with fixed capacity C.
$\theta \leftarrow \mathbf{0}_{1\times W}$ ▷ Vector that stores request latencies
Form probability map $\mathbb{D}$ where each request i has $\Pr[i] = \frac{1}{\theta[i]+\mu}$ where μ is a tiny positive constant.

Part 1 – When to checkpoint

procedure OnContainerStart(container K, starting request number R)
    $\mathbf{v} \leftarrow$ re-computed $\mathbb{D}$
    $\mathbf{v} \leftarrow \mathbf{v}[R, R+\beta]$ ▷ Clipped distribution
    Draw r' from interval $[R, R+\beta]$ using weights $\mathbf{v}$
    Checkpoint container K when it reaches request r'
    Save the snapshot in pool **P**
end procedure

Part 2 – Which snapshot to use

function GetSnapshotWeights
    $\mathbf{v} \leftarrow$ re-computed $\mathbb{D}$
    $\mathbf{w} \leftarrow \mathbf{0}_{1\times C}$ ▷ Snapshot weight vector
    for each snapshot K at index i in pool **P** taken at request number R_0 **do**
        $w[i] \leftarrow \frac{1}{\beta}\sum_{i=R_0}^{R_0+\beta} v[i]$ ▷ Average lifetime weight
    end for
    return w
end function

procedure OnContainerInit ▷ Each new container
  w ← **softmax** (GetSnapshotWeights)
  Draw snapshot K from pool **P** with snapshot weights given by **w**
  Resume from snapshot K
end procedure

Part 3 – How to update weights

procedure OnRequest(request number R, latency L)
  if $\theta[R]$ is 0 **then**
    $\theta[R] \leftarrow L$ ▷ Initialize with latency
  else
    $\theta[R] \leftarrow \alpha \cdot L + (1 - \alpha) \cdot \theta[R]$ ▷ Exponentially-weighted update
  end if
end procedure

Part 4 – Eviction policy for snapshot pool

procedure OnCapacityReached
  w ← GetSnapshotWeights
  P' ← top p% of snapshots in **P** using weights **w**
  Add γ% of snapshots in **P** chosen uniformly at random to **P'**
  P ← **P'** ▷ Discard remaining snapshots
end procedure

B Artifact Appendix

B.1 Abstract

Pronghorn is a snapshot orchestrator for serverless platforms. Its primary goal is to accelerate the execution of serverless functions.

B.2 Description & Requirements

B.2.1 How to access. Pronghorn can be accessed via the publicly hosted repository at the following URL: https://github.com/rssys/pronghorn-artifact/. It can also be accessed at the Zenodo artifact link here.

B.2.2 Hardware dependencies. To run Pronghorn effectively, it is recommended to run on an x86_64 machine with at least 8 cores, 32 GB of memory, and 256 GB of storage.

B.2.3 Software dependencies. To set up and use Pronghorn, you will need specific software dependencies. These include a Linux operating system (Ubuntu 22.04 is recommended), Docker Engine v20.10.12+ (version 20.10.12 or higher), and Kubernetes Server v1.21.1+. Additionally, ensure that you have several tools installed to facilitate the setup process. These tools include Multipass v1.12.2+, Arkade v0.8.28+, Helm v3.5.2+, Kubectl v1.2.22+, k3sup v0.11.3+, and faas-cli v0.14.2+.

B.2.4 Benchmarks. The descriptions of the benchmarks bundled along with the artifact can be found in Table 3 of the paper.

B.3 Set-up

To set up Pronghorn, follow the `README` in the Github project above or the Zenodo snapshot.

B.4 Evaluation workflow

B.4.1 Major Claims.

- *(C1): Pronghorn's orchestration strategy significantly improves request latencies for compute-bound workloads over state-of-the-art.*
- *(C2): Pronghorn's request-centric strategy consistently outperforms or matches the state-of-the-art across varying request traffic patterns, which is crucial for deployment in production.*
- *(C3): Pronghorn minimizes checkpointing overhead by undergoing an exploration period, resulting in brief container downtime. This downtime is similar to state-of-the-art systems and is not directly experienced by users.*
- *(C4): Pronghorn's request-centric policy converges to an optimal snapshot in a small number of requests, allowing cloud providers to stop further checkpointing after a specific number of invocations.*

B.4.2 Experiments. *Experiment (E1): [30 human-minutes + 48 compute-hour]: In this experiment, we conduct a comprehensive evaluation encompassing all benchmarks, strategies, and eviction rates, as presented in Section 5 of the paper.*

[Preparation] The Pronghorn setup process automatically generates the necessary configurations for this experiment. No additional steps are necessary.

[Execution] To initiate the experiment, simply execute the `run.sh` *script located in the root directory with the 'evaluation' argument. Given the extended duration of this experiment, we advise running the command in the background using a mechanism such as* `nohup ./run.sh evaluation &` *to ensure uninterrupted execution.*

[Results] The results will be created as CSV files in the `results/` *directory. The* `Evaluation.ipynb` *plotting script provided in the* `figures/` *directory can be used to interactively create Figure 4 and Figure 5 of the paper.*

Experiment (E2): [1 human-hour + 1 compute-hour]: This experiment enables the assessment of the system's checkpoint and restore overhead.

[Preparation] Deploy any function using `faas-cli deploy –image=USER/workload –name=workload`

[Execution] Copy the script `cost-analysis/table4.py` *to the pod created by OpenFaaS. Next, attach to the pod using* `kubectl exec -it $pod_name – /bin/sh` *and run the script within the pod. Copy the JSON emitted by the program to a file that can be used for the analysis.*

[Results] If required, this can be done for all functions. However, for convenience, a result JSON file has been attached from our evaluation run. The results provide the numbers for the checkpoint, restore, and snapshot overheads presented in Table 4.

Experiment (E3): [10 human-minutes + 10 compute-minutes]: This experiment quantifies the number of requests needed for Pronghorn to reach an optimal snapshot state.

[Preparation] The evaluation run will produce the necessary inputs for this experiment.

[Execution] To compute and display the results, simply execute the `cost-analysis/evaluation_cost.ipynb` *notebook.*

[Results] The output obtained from the notebook can be directly compared with Table 4 of the paper.

Experiment (E4): [10 human-minutes + 10 compute-minutes]: This experiment allows evaluating the storage and network bandwidth usage of Pronghorn.

[Preparation] The data collected for E2 will produce the necessary inputs for this experiment.

[Execution] To compute and display the results, simply run the `cost-analysis/table_5.py` *notebook.*

[Results] The output obtained from the notebook can be directly compared with Table 5 of the paper.