λ-TRIM: Optimizing Function Initialization in Serverless Applications With Cost-driven Debloating

Xuting Liu*
University of Pennsylvania
Philadelphia, PA, USA
xutingl@seas.upenn.edu

Spyros Pavlatos*
University of Pennsylvania
Philadelphia, PA, USA
pavlatos@seas.upenn.edu

Yuhao Liu
University of Pennsylvania
Philadelphia, PA, USA
liuyuhao@seas.upenn.edu

Vincent Liu
University of Pennsylvania
Philadelphia, PA, USA
liuv@seas.upenn.edu

Abstract

In this paper, we focus on an often-overlooked component of serverless application cold starts: monetary costs and Function Initialization. Traditionally considered the user's responsibility, Function Initialization is billable and accounts for more than 50% of the monetary cost associated with cold starts in real-world machine-learning applications.

We introduce λ-TRIM, a system that optimizes Python serverless applications by eliminating redundant code while maintaining correctness. To maximize cost savings, λ-TRIM leverages the typical serverless pricing model to prioritize modules that significantly impact latency and memory usage. λ-TRIM features an automated pipeline comprising a static analyzer, a profiler specialized for the serverless pricing model, and a debloater. The optimized application can be directly deployed on serverless platforms, leading to substantial reductions in both latency and cost for cold starts.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Keywords: Serverless computing, cold starts, debloating

ACM Reference Format:
Xuting Liu*, Spyros Pavlatos*, Yuhao Liu, and Vincent Liu. 2025. **λ-TRIM**: Optimizing Function Initialization in Serverless Applications With Cost-driven Debloating. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3676642.3736129

*Denotes equal contribution and alphabetical ordering.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1080-3/2025/03
https://doi.org/10.1145/3676642.3736129

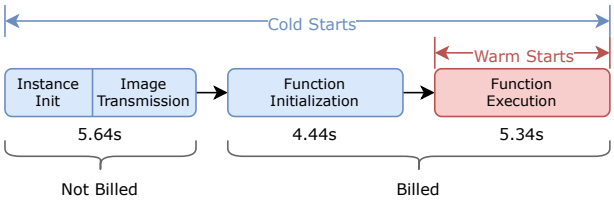

Figure 1. A typical breakdown of cold and warm starts for a PyTorch ResNet invocation and the average execution time of each phase. Function Initialization is responsible for up to 29% of total latency and 45% of the total bill in a cold start.

1 Introduction

Serverless computing is an increasingly popular paradigm that allows clients to run their applications on cloud providers without worrying about the prosaic but complex tasks of provisioning, scaling, and maintaining VMs or containers [26]. Under the serverless abstraction, users provide a function to the cloud provider, which handles everything else automatically. The user is then billed per-MB of provisioned memory, per-millisecond[1] of request processing time; they only pay for what they use. Idle time is free.

The lifecycle of a serverless function, depicted in Figure 1, consists of three phases: instance/runtime setup time, Function Initialization, and Function Execution. User logic is primarily contained within the Function Execution stage.

In this work, we call attention to, quantify, and address the fact that, among these stages, Function Initialization plays an outsized role in the overhead of serverless computing. Even though this stage is not executed for every request, its costs can be substantial, both in latency and resources, the latter of which manifests as higher monetary costs to users.

One way that Function Initialization adds overhead is during so-called *cold starts*, where the cloud platform is forced to initialize—in the critical path—a new serverless instance in response to an incoming request and insufficient existing capacity. This contrasts warm starts, where the cloud provider can reuse a previously initialized VM/container. As many others have also noted [23, 30, 45], these cold starts can be common for some applications, and their latency penalty can be substantial, accounting for up to an 80% increase in latency compared to warm function execution. Within

[1]AWS Lambda pricing granularity. GCP rounds up to the nearest 100 ms, and Azure rounds up to the nearest 1 s.

cold starts, Function Initialization is of particular concern given trends toward more/heftier libraries (e.g., ML, scientific computing, and image/video processing) [35] and scale-out architectures that lead to very bursty work- loads [21].

In addition to latency, Function Initialization also differs from other cold-start components in that cloud providers typically bill users for the time, as shown in Figure 1[2]. There are good reasons for this pricing strategy: not billing for initialization could result in perverse incentives, e.g., users breaking up expensive computations into a sequence of n functions where each does $\frac{1}{n}$ of the computation, and stores/loads partial results—the billable work would be a no-op. The result, however, is that for the ResNet application of Figure 1, initialization can be responsible for up to 45% of the total bill.

Finally, the overheads of Function Initialization can persist even after the instance has been warmed, as large libraries and data structures instantiated during initialization will continue to occupy memory even if they are never used, consuming resources that again manifest as high monetary costs for every user request.

Prior work has looked at parts of this problem, especially through the lens of optimizing cold start latency, e.g., through OS improvements [6, 11, 20], better function scheduling [13, 32, 44, 45, 55], checkpoint/restore techniques [20, 46], and others. Unfortunately, a fixation on just one part of the problem, like cold start latency, can lead to tradeoffs on the other aspects (e.g., the resource costs of checkpoint/restore techniques that are detailed in Section 8.6). The few approaches that target both latency and monetary cost are largely manual (e.g., developing lightweight libraries [47] or relatively simple methods to refactor the application [30, 48, 50]).

This paper presents λ-trim, a system that optimizes Python serverless applications by profiling and eliminating unnecessary initialization operations. λ-trim's optimizations minimize the latency of cold starts and the monetary cost of both cold *and* warm executions. λ-trim operates entirely as a pre-processing step at the application level—its output is an optimized serverless application with a shorter Function Initialization phase and less memory footprint. It is, thus, immediately deployable and remains compatible with system-level efforts toward cold start optimization.

Under the hood, λ-trim leverages a well-known technique from the programming languages and software engineering fields, Delta Debugging (DD) [53]. DD takes a divide-and-conquer approach to finding the largest subset of the code base that it can remove while producing correct results. In each iteration, DD splits the program into multiple subsections and examines each subsection to determine whether it is necessary for correct execution, repeating the process until it reaches a minimal configuration.

[2]As we discuss in Section 2.1, some functions' initialization is complimentary, but this is not true in general.

Unfortunately, applying DD to every line of code in the serverless function and its dependencies is impractical. A contribution of λ-trim is, thus, to leverage typical serverless pricing models (via an estimate of marginal monetary cost) to enable efficient targeting of DD-based debloating. λ-trim's system architecture features an automated pipeline encompassing a static analyzer, serverless cost profiler, and DD-based debloater. Our key contributions are as follows:

- We analyze the latency and cost breakdown and find the initialization phase to be a significant overhead in many serverless Python applications.
- We demonstrate—empirically—the substantial redundancy in those initialization phases with λ-trim.
- As part of λ-trim, we introduce the first practical advanced Python debloater using a workflow specialized for serverless platforms and their unique pricing models.
- We evaluate λ-trim on real serverless applications and reduce monetary costs by an average of ~20% (cutting many applications' costs by >50%) while also improving E2E latency by up to 2× and memory usage by up to 42%.

2 Background and Motivation

In traditional cloud computing models, users are responsible for a wide range of system administration tasks not directly related to their application logic, e.g., provisioning a batch of VMs, specifying their resource profiles, deploying dependencies, scaling the instance up and down with the workload, and monitoring the application as it runs, among others.

Serverless computing is an alternative that promises to free users from all the above concerns. Instead, users simply supply the cloud provider with a function containing their application's logic (commonly known as the *serverless function* or *lambda*). The provider handles the rest.

This abstraction offers many benefits, including: (1) users are relieved from the need to manage servers, (2) resources are automatically scaled based on demand, and (3) users are billed for only the resources they use and no more. This paradigm has proven popular, with all large cloud providers offering a range of options for serverless execution (e.g., AWS Lambda, GCP Cloud Run functions, and Azure Functions).

The workflow for developing such applications is straightforward. Users write a function in their preferred programming language, package the function code and any necessary libraries into a suitable format (e.g., a container image or ZIP file), and then upload it to the serverless platform. The serverless platform manages provisioning and execution.

2.1 The Anatomy and Pricing of Lambda Execution

Lambdas are executed on-demand, invoked by a predefined set of triggers such as incoming HTTP requests, event triggers (e.g., a file upload or monitoring alert), and scheduled timers. Serverless platforms ensure automatic scaling by dynamically launching new instances to handle invocations and shutting them down when they are no longer needed.

Cold/warm starts. As a result of dynamic scaling, when an incoming request is received after a period of inactivity or as part of a burst that exceeds the capacity of the currently deployed instances, the invocation incurs what is known as a *cold start*. In a cold start, the cloud provider must initialize a new VM, including loading the runtime environment, loading dependencies, initializing the application code, and establishing connections (e.g., to databases). Partly because of its advantage in cold-start latency (on top of their ease of use and popularity), interpreted languages like Python remain the most popular choices for serverless runtimes [18].

Once a serverless instance is initialized, the instance remains active for a keep-alive period that is reset on a new request to the instance in question. In AWS Lambda, the keep-alive period is up to ~45–60 min, but potentially much less depending on the size of the instance and resource availability [31]; in GCP, the period is ≤15 min. If another request arrives during this period and the instance is not already processing a request, it can execute the new request without repeating initialization, resulting in a *warm start*.

Pricing. Most serverless platforms employ a pricing model based on both the memory usage of the application and the duration for which the serverless function runs. Allocation of other resources like CPU and network bandwidth depends on the cloud provider and specific pricing plan. AWS, for instance, allocates both resources proportionately to the memory footprint, with additional vCPUs assigned at designated memory allocation breakpoints. Azure allocates a fixed CPU and memory (100 ACU and 1.5 GB) budget per function instance, with additional configuration options for premium hosting plans, while GCP allows independent configuration of CPU/memory in their v2 API.

For a given invocation, the pricing is thus primarily determined by the footprint and duration of the function. For example, AWS Lambda charges users as follows:

$$C = \textit{Configured Memory} \times \textit{Billed Duration} \times \textit{Unit Price} \quad (1)$$

In AWS Lambda, billing is computed in 1 ms increments [10], and memory configurations range from 128 MB to 10 GB. Configuring the memory too large is a waste of resources and money. Configuring it too small would result in memory swapping, which can degrade server performance. The billed duration would significantly increase in this case, hurting both latency and cost. As a result, the optimal configuration should be above the application's peak memory footprint.

As shown in Figure 1, the billed duration of cold starts generally includes both Function Initialization and Function Execution[3]. In short, everything involved with executing the container image uploaded by the user is billed. In contrast, the cloud platform is responsible for the preparation process, including setting up the physical server and downloading the application image from a storage server. Time spent in this stage is reflected in the E2E latency but will not appear on the bill. As such, serverless platforms are strongly incentivized to optimize this phase but are much less motivated to help users reduce Function Initialization costs.

[3]The exception is functions on AWS Lambda that use zipped code on managed runtimes and initialize in <10 s [49]. Initialization is not charged for these functions, but AWS imposes size restrictions on the zipped code that are impractical for the types of applications we consider here.

Application	External modules	Size (MB)	Time (s) Import	Exec	E2E
From *FaaSLight* [30]					
huggingface	`torch`, `transformers`	799.38	5.52	0.86	10.12
image-resize	`boto3`, `wand.image`	102.05	0.42	0.95	1.88
lightgbm	`lightgbm`, `numpy`	120.22	0.57	0.04	1.14
lxml	`requests`, `lxml`	58.01	0.24	0.39	1.12
scikit	`sklearn`	177.01	0.30	0.01	1.93
skimage	`skimage`	155.37	1.87	0.10	2.76
tensorflow	`tensorflow`, `numpy`	586.13	4.53	0.04	5.33
wine	`numpy`, `pandas`, `sklearn`, `boto3`	271.01	1.96	0.29	2.81
From *RainbowCake* [51]					
dna-visualization	`squiggle`	57.01	0.18	0.02	0.72
ffmpeg	`ffmpeg`	297.00	0.06	2.50	3.07
igraph	`igraph`	40.00	0.09	0.01	0.59
markdown	`markdown`	32.21	0.04	0.03	0.54
resnet	`numpy`, `torch`, `PIL`	742.56	6.30	5.30	11.71
textblob	`textblob`	104.00	0.42	0.38	1.28
New Applications					
chdb-olap	`chdb`	293.64	1.01	0.08	1.77
epub-pdf	`reportlab`, `pptx`, `docx`, `boto3`	143.68	0.62	1.43	2.54
jsym	`sympy`	83.01	0.56	0.31	1.36
pandas	`numpy`, `pandas`	114.27	0.67	0.01	1.19
qiskit-nature	`qiskit_nature`	281.15	1.96	0.49	3.05
shapely-numpy	`numpy`, `shapely`	58.42	0.20	0.01	0.71
spacy	`spacy`, `boto3`	202.00	2.06	0.02	2.60

Table 1. Benchmarked applications

2.2 Function Initialization in the Wild

To investigate the overheads of Function Initialization, we study real serverless applications. We conduct our experiments on AWS Lambda using a Python 3.10 runtime.

2.2.1 Benchmarked Applications

We collect a comprehensive set of serverless applications by constructing a union of applications used in other work, namely FaaSLight [30] and RainbowCake [51]. To augment this set of applications, for some of the 20 largest and most popular packages in PyPI [34] (excluding nightly versions), we select a representative, real-world, and open-source serverless application found via GitHub search and add it to the union as well. Finally, we remove any repetitive applications that implement similar tasks. We prioritize applications that are the most recent and have clear instructions to run. For example, a machine learning image classification application that uses PyTorch appears in all three of our sources, and we keep the RainbowCake version.

Our final serverless application set consists of 21 real-world applications from FaaSLight (8), RainbowCake (7), and PyPI (6). The FaaSLight and RainbowCake benchmarks contain 15 and 10 Python applications, respectively.

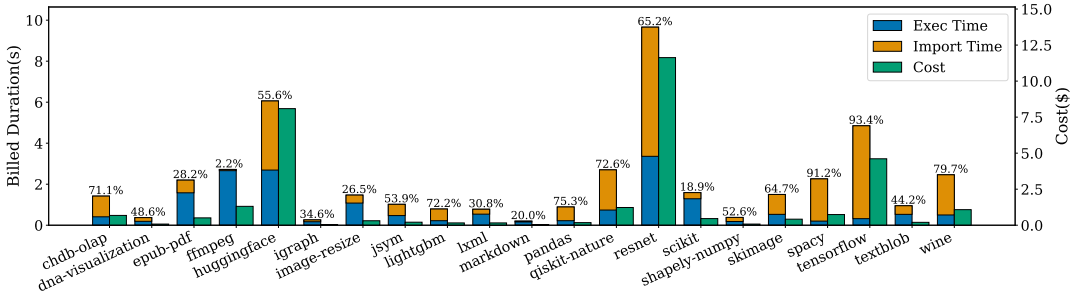

Figure 2. Billed duration (left bar) and monetary cost (right bar) of cold starts for each serverless application. The billed duration, priced for 100K invocations, is further divided into Function Initialization (Import) and Function Execution (Exec) time. The label on the bar is the percentage of import time out of the total billed duration.

Algorithm 1 The generic Delta Debugging algorithm.

Require: Program P, Oracle O
Ensure: 1-minimal program P' s.t. $O(P') = \text{T}$

$A \leftarrow$ list of program components of P
$n \leftarrow 2$
repeat
  $\langle a_1, \ldots, a_n \rangle \leftarrow$ split A into n partitions
  if $\exists i.\ O(a_i) = \text{T}$ **then**
    $\langle A, n \rangle \leftarrow \langle a_i, 2 \rangle$
  else if $\exists i.\ O(A \setminus a_i) = \text{T}$ **then**
    $\langle A, n \rangle \leftarrow \langle A \setminus a_i, n - 1 \rangle$
  else
    $\langle A, n \rangle \leftarrow \langle A, 2n \rangle$
  end if
until $n \leq |A|$
return 1-minimal P'

2.2.2 Metrics

We invoke the above collection of serverless applications and collect their latency and monetary cost. We focus on cold starts, which include all phases of execution.

Latency. End-to-end cold start latency (E2E) is the duration between the issue of a user request and the response from AWS Lambda. E2E latency can be further broken down into the four phases in Figure 1 [30], but pay particular attention to the two phases under the control of users: Function Initialization and Function Execution. We collect Function Initialization (Import) latency by instrumenting the benchmarked applications with recorded timestamps before and after the Lambda initialization code block. AWS directly reports the Function Execution latency. We report Import, Execution, and E2E latency of each application in Table 1.

Monetary cost. As discussed in Section 2.1, AWS Lambda charges each invocation based on both billed duration and configured memory. Although it is possible to configure 128 MB to 10 GB memory for any serverless application on AWS Lambda, for the best cost-effectiveness, the memory should be set proportional to the memory footprint [9]. As a lower bound, we report the measured maximum memory footprint of the application for a single request—in practice, there will be some additional headroom. We set memory to 128 MB in cost calculation if the measured memory is less. We report the monetary cost for 100K invocations, calculated using the unit price of $0.0000162109 per GB per second [10].

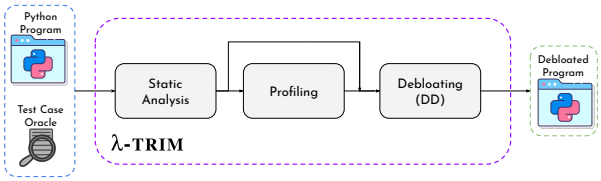

Figure 3. Architecture of λ-TRIM, which includes three components: a static analysis phase, a profiler, and a debloater.

2.2.3 The Overheads of Function Initialization

The results of our measurement study are shown in Figure 2. Across all applications, we find that Function Initialization time accounts for a disproportionate fraction of cold start latency. Especially when considered as a fraction of billed duration, initialization time is often greater than the actual function execution time, with the worst offenders (i.e., **spacy** and **tensorflow**) spending >90% of their billed duration on initialization tasks. The median share for initialization tasks is 53.75%, but the proportion is generally higher for larger applications (e.g., **resnet** and **huggingface**, which spend 62% and 65% of their billed duration on imports, respectively).

We note that the actual impact of Function Initialization on monetary cost is much higher than the contribution to latency reported here since (as we will see in Section 8.1) the initialization tasks also lead to additional memory allocations that must be carried through the life of the function. Thus, when a serverless application—typically consisting of a single, focused task [50]—imports a large library with modules that will never be used in the execution phase (e.g., importing the forward pass of a neural network model but getting a more general definition of the model), the impact on monetary costs is outsized.

3 Related Work and Approach

3.1 Related Work in Serverless Optimization

The majority of work in optimizing serverless functions focuses on cold start latencies.

One approach is to optimize the serverless infrastructure itself, e.g., through many of the techniques cited in Section 1 such as OS improvements [6, 11, 20], optimized function scheduling [13, 32, 44, 45, 55], checkpoint/restore [20, 46], caching [14, 15, 23, 51], provisioned concurrency [2], pre-warming [38, 45] and memory/resource sharing [28, 29, 42]. Of these, checkpoint/restore (C/R) is of particular note as it directly accelerates the Function Initialization phase. Checkpointing involves saving the runtime state of a serverless function after initialization, including memory, execution context, intermediate computations, or even a snapshot of the whole VM. During a cold start, the serverless platform can restore from the checkpoint instead of starting from scratch. Unfortunately, as we will see in Section 8.6, they come with a cost, and that cost is being exacerbated by the same trends that motivate this work: trends toward more/heftier libraries and scale-out architectures—aggregate checkpoint sizes grow in both cases.

More generally, the drawback of framework-level optimizations is that they require privileged access to the underlying serverless infrastructure. This means that users cannot use them to speed up their applications unless the optimizations are adopted by the serverless vendors.

Application-level optimizations do not require privileged access to the infrastructure, but existing solutions all rely exclusively on static analysis and/or human intervention, which limits their scope and efficacy. For instance, LibProf [47], while helpful in providing advice, requires a human to design and implement the optimizations. Function fusion [43, 48] is automatic and reduces cold start frequency, but at the cost of the performance of the cold starts that do execute. Finally, static analysis techniques like *FaaSLight* [30] can be automated but requires extensive manual annotation to achieve good performance [1]. *FaaSLight* additionally retrieves the original code as a safeguard, yielding additional overheads.

More generally, cold-start latency is only one of the overheads of Function Initialization, and the others—the monetary costs of cold and warm starts—are arguably the more important metrics for many users. In fact, a fixation on latency can make the other axes worse. For example, C/R comes at the cost of resource overheads for storing and restoring state, the cost of which (as we show in Section 8.6) often overwhelms the cost of actually running the function.

3.2 A Path Forward: Delta Debugging (DD)

Our work, λ-trim, borrows from a technique called Delta Debugging (DD). DD is a general approach to program minimization that has been used for tasks from isolating faulty/insecure code to tracking down configuration issues. Initially, DD was used as a tool to minimize crashing programming inputs [52, 53], but in recent years has been adapted to perform program debloating [25]. For the debloating problem, DD takes as input:

- a program P that can be decomposed into a list, A, of components, e.g., statements, functions, tokens, etc.
- an oracle O that returns T if the program fulfills a desired property and F otherwise.

and tries to find a minimal program with respect to the number of components such that the oracle returns T. Note, however, that finding the minimum number of components is NP-complete [53] and impractical for any reasonably sized problem. Instead, DD targets a different property: *1-minimality*. Essentially, a program P^* is called 1-minimal if it satisfies the oracle, and removing any single component from P^* leads the oracle to return F. These local minima are sufficient for most practical cases.

The DD algorithm. The general DD algorithm, as introduced in [25], is shown in Algorithm 1. The algorithm uses a divide-and-conquer approach that begins by setting the solution candidate, A, to the entire program, and the number of partitions, n, to 2.

In each iteration of the algorithm, we split the current solution candidate A into n partitions, $\{a_1, \ldots, a_n\}$. For each partition a_i, we query the oracle to check if it returns T, i.e., partition a_i satisfies the target property. If it does, we eliminate the remainder of the program from consideration and repeat the process with a solution candidate of $A \leftarrow a_i$ and a partition granularity of $n \leftarrow 2$.

If, on the other hand, none of the partitions pass the oracle test, we also test each of their complements, i.e., for partition a_i, we test $A \setminus a_i$. If a complement passes the oracle test, we again narrow down our solution candidate, but here we set the new granularity to $n \leftarrow n - 1$.

Finally, if neither the partitions nor their complements pass the oracle test, the algorithm doubles the granularity $n \leftarrow 2n$ and repeats the process. The algorithm terminates if the maximum granularity is exceeded, i.e., $n > |A|$, then we return the current solution A as the minimal program P^*.

4 Design Overview

λ-trim reduces the overheads of Function Initialization in serverless functions. While our prototype implementation targets Python, we note that our techniques can be applied in a very similar way to other interpreted languages like Javascript[4] (see Section 6.1) and can be extended to compiled languages as demonstrated by prior applications of DD [25, 53], albeit at the cost of compilation overheads during the debloating process. In any case, λ-trim's approach is to—through pure application pre-processing—remove code from applications' dependency chains that are not needed for the application to run.

At the core of our approach is the DD technique described above; however, we emphasize that a naive application of DD is impractical. In fact, to the best of our knowledge, DD

[4] ~70% of all serverless applications are written in Python or Javascript [17].

and other, more advanced debloating techniques have never been applied to interpreted languages like Python, despite its immense popularity [19].

There are several reasons why debloating Python is a challenging task. First, Python allows for dynamic imports, as modules can be loaded at runtime. As a result, a static approach would need to be over-conservative so that it does not remove any module that *might* be imported during the actual program execution. Second, as discussed in Section 2, today's applications depend on large third-party libraries that are intractable to fully debloat, even with powerful algorithms like DD; it is precisely these large libraries that are most important to trim down.

System design. To tackle the above challenges, we propose λ-trim. λ-trim utilizes a pipeline consisting of a static analyzer (Section 5.1), a profiler (Section 5.2), and a debloater (Section 5.3) to remove redundant code from serverless applications. The architecture of λ-trim is shown in Figure 3. λ-trim accepts as input:

1. A Lambda-compatible Python program and associated deployment image.
2. An oracle specification, i.e., a set of inputs to the Python program for which the debloated program needs to give the same output as the original.

The input program is passed through the static analyzer, which identifies the external modules that the application imports. λ-trim then uses a billing cost-based model to profile these modules and restricts the debloating process to the modules that would affect the application the most when the application is deployed on the serverless platform. Finally, λ-trim debloats this set of imported modules using DD and produces optimized code for these modules as output.

Benefits. By stripping away excess, λ-trim helps reduce memory usage, execution time, and as a result, monetary costs in a way that is both backward compatible and complementary to other cold start optimizations; its output can be deployed to AWS Lambda directly with no modification to the application or underlying infrastructure.

Further, although λ-trim is aggressive in its removal of functions, classes, and module imports, the oracle specification provides strong guarantees against potential inputs.

5 λ-trim Workflow

In this section, we detail the components of λ-trim and their responsibilities in optimizing an application.

Program Inputs. Serverless applications consist of two parts: (a) initialization code and (b) a designated function handler. Initialization code consists of library loading and environment setup. For instance, in Python applications, this may include `imports`, definitions of helper functions, establishing connections with databases or other services, etc. All of these execute once per function instance as part of the cold start process. The handler, on the other hand, is the entry point that takes a request and processes it; the serverless platform calls into this handler as new requests arrive.

```python
# Initialization code
import boto3

session = boto3.Session(
    aws_access_key_id=..., aws_secret_access_key=...
)

# Lambda function
def handler_name(event, context):
    ...
    return some_value
```

Figure 4. Example of a serverless application that establishes a `boto3` session to manage and interact with AWS services.

A minimal example that utilizes AWS SDK for Python is given in Figure 4. The entry point to the application is the `handler` function, which takes as arguments an `event` and a `context`. An event is a `JSON` formatted object that contains data for the lambda function to process, while the `context` object provides information about the invocation, function, and runtime environment [8]. Code outside of the `handler` counts as the Function Initialization phase, which in this case includes an `import` and `boto3` session setup.

λ-trim expects two user inputs. The first is an application in the above format, i.e., a Python program with a lambda handler. The second is the oracle specification, i.e., `JSON` file containing the input test cases that λ-trim will use to ensure correctness. Each test must contain an `event` and a `context`.

5.1 Static Analyzer

The first step in λ-trim is to obtain information about the input program and potential candidates for debloating.

Specifically, λ-trim executes a single pass over the Abstract Syntax Tree (AST) of the program to identify all imported modules and then employs the state-of-the-art Python static analyzer `PyCG` [41] to obtain the call graph of the input program. The call graph gives information about the attributes of the modules that are *definitely* accessed by the application. These attributes can safely be excluded from the DD process, which speeds up the debloating phase. The final list of modules is then passed to the debloater.

5.2 Profiler

While, in principle, a debloater could examine all of the modules imported by the application (minus those that are definitely accessed), modern serverless applications—particularly those that might benefit from λ-trim—are large enough to render such an approach intractable. Instead, λ-trim leverages a cost-guided profiling step that helps the debloating process to prioritize modules with the most potential impact.

Top-K ranking of the *marginal monetary cost*. While predicting the potential execution time and memory footprint savings of module removal is difficult in general (equivalent

to solving the halting problem), we find *marginal monetary cost* to be sufficient to identify a set of potential candidates for the debloater. We define the marginal monetary cost as:

$$\textit{Marginal Monetary Cost} = TM - (T - t)(M - m) \quad (2)$$

where t and m are the marginal import time and the memory footprint of modules and all their submodules, respectively, and where T and M are their sums over all imported modules.

All four values (t, m, T, and M) are measured by patching Python's import machinery. In particular, we modify Python's module loader by inserting time and memory measurements before each module execution. The t and m of a module are equal to the difference in T and M before and after the execution of that particular module.

While an imperfect solution, we find that the above heuristic avoids most pathological application structures. For example, a strawman that only considers execution time might pick a module that is slow but does not require memory (usually a result of an un-trimmable loop in the initialization).

5.3 Debloater

λ-trim implements a general debloater for Python applications that uses DD as the underlying program minimization algorithm. The top-K modules from the profiler are fed into the debloater. The debloater uses the output of PyCG to mark the necessary attributes and proceeds to debloat each module with the rest of the module's attributes. The eventual output of the debloater is a set of optimized modules.

In each iteration of DD, the debloater modifies the module and tests the output of the modified program given each test case of the oracle specification as input. In most cases, just ensuring the matching of standard output is sufficient; however, extensions to other observable effects found in serverless applications is straightforward.

In particular, the stateless nature of serverless applications means that local side effects (e.g., file system changes) can be ignored. Rather, serverless state and side effects are comprised of external calls to remote services and other serverless functions—validating these types of functions involves intercepting such operations and checking for equivalence.

5.4 Deployment With Fallbacks

Finally, the optimized program is packaged into a container image that is deployed to the serverless platform. λ-trim, like similar program analysis techniques, relies on the oracle as a high-level specification and assumes that users will provide a strong enough set of test cases to ensure correctness. Even so, λ-trim provides a fallback mechanism that can correctly handle cases where λ-trim removes a necessary attribute. Specifically, if an input ever accesses a deleted attribute, it will trigger an `AttributeError`. λ-trim wraps the debloated function to catch these errors and, when detected, invoke the original function as an independent serverless instance. The return value of the wrapper is the response from the original function and a notification about the failing input.

During normal operation, the overhead of this wrapper is negligible. The tradeoff is that the overheads of actually triggering the fallback can be high (see Section 8.7). That said, the fallback mechanism is a safety net that should be executed very rarely and, when it is triggered, should alert the user to re-run λ-trim with an updated oracle set.

Note that re-execution of a non-idempotent function may cause inconsistencies and side effects. However, these types of re-executions already exist even without λ-trim, so non-idempotent applications should already be handling these cases (e.g., using a framework like Beldi [54]).

More broadly, we note that there are well-known techniques to assist users in creating oracle sets for these types of tools. For example, one common and relatively robust approach is running a fuzzer against the optimized program. If the fuzzer finds a failing input, then the user can add the input to the oracle set and rerun λ-trim.

6 The Debloating Process

6.1 Tailoring DD for Serverless Python Applications

A critical design decision in λ-trim is to identify the appropriate debloating granularity not only based on Python's semantics but also based on the potential speed-up of the loading time when the application is deployed on AWS Lambda.

At a high level, everything in Python is treated as an object. As such, modules are also Python objects that wrap around a dictionary that maps names to other objects. This dictionary defines the namespace of the module, i.e., the attributes of the module that we can access after we import it.

When a Python module is imported, all the statements in the module execute in program order. Python's import machinery constructs the namespace of the module on the basis of each statement. For example, the statement `import module` creates a module object for `module` and adds it to the namespace. Similarly, the definition of functions and classes creates the corresponding `function` and `class` objects.

Attributes are, thus, the building blocks of a module, and we see an opportunity to run DD with this granularity instead of at the granularity of statements. Compared to statement granularity, attribute granularity is coarser with respect to `function` and `class` definitions, the same for `import` statements, but more fine-grained for `from module import attr` statements, since `attr` can be a list of attributes.

To minimize the overheads of Function Initialization, we use DD with attribute granularity to debloat imported modules. By doing so, we not only eliminate `function` and `class` definitions and the costly `import` statements but also remove unused module attributes from `from import` statements, thus reducing the memory footprint of the created module objects. With statement granularity, we cannot remove specific attributes, as it removes all or none of them.

```python
import torch

x = torch.tensor([1.0, 2.0])
y = torch.tensor([3.0, 4.0])
z = view(torch.add(x, y), 2, 1)
model = torch.nn.Linear(2, 1)
model.weights = torch.tensor([[1.0], [2.0]])
model.bias = torch.tensor([3.0])

print(model(z))
```

Figure 5. Sample application that uses a simplified `torch`.

Generalizability. It is worth discussing the generalizability of the above techniques to other interpreted languages like Javascript (JS), though a full exploration of the design is out of scope. JS offers a similar import model as Python; one can import specific exports from another module, similar to the `from import` statement of Python. Thus, DD can be adjusted in a straightforward way to JS modules. An additional complexity of JS is the wider range of module namespaces, which include URLs. To handle this, one can resolve namespaces statically and then proceed to DD.

6.2 Running Example

To illustrate our implementation of DD for Python programs, we will consider a simplified version of the `torch` module:

$$A = \left\{ \begin{array}{lll} \texttt{torch.tensor}, & \texttt{torch.add}, & \texttt{torch.view}, \\ \texttt{torch.nn.Linear}, & \texttt{torch.nn.MSELoss}, & \texttt{torch.optim.SGD} \end{array} \right\}$$

where `torch.tensor` is a tensor class and `torch.add` and `torch.view` are two tensor operations. `torch.nn.Linear` is a Neural Network layer from the `torch.nn` submodule. Finally, `torch.nn.MSELoss` and `torch.optim.SGD` are utilities for optimizing Neural Networks.

We import the `torch` module in the simple application shown in Figure 5. This application does not make use of `torch.nn.MSELoss` and `torch.optim.SGD`. Assuming that none of the other four attributes depend on them, DD will remove the redundant attributes from the `torch` module through the process shown in Figure 6.

After DD correctly identifies the redundancy of the two attributes and removes them from the library, the resulting module initialization code is shown in Figure 7. The debloated library now consists of:

$$A^* = \left\{ \begin{array}{ll} \texttt{torch.tensor}, & \texttt{torch.add}, \\ \texttt{torch.view}, & \texttt{torch.nn.Linear} \end{array} \right\}$$

The debloated library omits the attribute `torch.nn.MSELoss` and skips the import of `torch.optim` entirely.

6.3 Profiling-driven Debloater

Implementing the above, the results of the static analysis and profiling phases are fed into an attribute-level DD process. All the magic attributes of the module (e.g. `__file__`) [22] are excluded from DD. In each iteration of the algorithm, the original `__init__.py` file is retrieved and then modified based on the attributes that DD currently tests. The

1	tensor	add	view	Linear	SGD	MSELoss	✓
2	tensor	add	view	Linear	SGD	MSELoss	✗
3	tensor	add	view	Linear	SGD	MSELoss	✗
4	tensor	add	view	Linear	SGD	MSELoss	✗
5	tensor	add	view	Linear	SGD	MSELoss	✗
6	tensor	add	view	Linear	SGD	MSELoss	✗
7	tensor	add	view	Linear	SGD	MSELoss	✗
8	tensor	add	view	Linear	SGD	MSELoss	✗
9	tensor	add	view	Linear	SGD	MSELoss	✓
10	tensor	add	view	Linear	SGD	MSELoss	✗
11	tensor	add	view	Linear	SGD	MSELoss	✗
12	tensor	add	view	Linear	SGD	MSELoss	✗
13	tensor	add	view	Linear	SGD	MSELoss	✗
14	tensor	add	view	Linear	SGD	MSELoss	✗
15	tensor	add	view	Linear	SGD	MSELoss	✗
16	tensor	add	view	Linear	SGD	MSELoss	✗

Figure 6. Visual walkthrough of the DD algorithm applied to the simplified `torch` library. Attributes with blue background are the ones under test in the current iteration. Note that in step 10, we halve the granularity twice since all sets for $n = 2$ have been tested in previous iterations.

modification is achieved with a single traversal of the AST. The modified `__init__.py` file is then copied back to the `site-packages` directory.

For each of the modules in the top-K of marginal monetary cost, the debloating process consists of the following steps:

1. The module is loaded in order to access its attributes.
2. The `__init__.py` file of the module is backed up so that it can be retrieved in every iteration of DD.
3. A set of potentially redundant attributes is constructed containing all the attributes of the module, except those that are contained in the output of `PyCG` and the magic attributes of the module.
4. Run the DD algorithm for the module. Note that only the set of potentially redundant attributes defined in Step 3 are considered; all other code is untouched.

7 Implementation

The λ-trim implementation comprises roughly 1.1k LoC of Python. The only third-party packages we use are `PyCG` [41] to extract the call graph of applications and `psutil` to measure the memory footprint of the imported modules. We have tested our implementation against Python 3.10. There are two implementation details that are important to note.

Module isolation. When a Python module is imported, it is cached by the interpreter to optimize subsequent imports. This caching, however, prevents us from conducting static analysis before the profiling phase, since modules need to be loaded to retrieve their AST. As a result, the Python interpreter would use the cached version of each module, leading to inaccurate measurements of the module's import time.

To address this, λ-trim imports modules in isolation. Specifically, a new process is spawned in both the static analysis and the profiling phase. A new process is also spawned for each run of DD for the top K module. By spawning a new

```python
from torch.nn import Linear, MSELoss
from torch.optim import SGD

class tensor():
    def __init__(self, ...):
        ...
def add(t1: tensor, t2: tensor) -> tensor:
    ...
def view(t: tensor, dim1: int, dim2: int) -> tensor:
    ...
```

(a) Original `torch` library.

```python
from torch.nn import Linear
pass

class tensor():
    def __init__(self, ...):
        ...
def add(t1: tensor, t2: tensor) -> tensor:
    ...
def view(t: tensor, dim1: int, dim2: int) -> tensor:
    ...
```

(b) Debloated `torch` library.

Figure 7. Simplified version of `torch` (a) before and (b) after debloating.

process for each phase, we provide each with its own address space, preventing modules from being cached across phases.

Deployment. λ-trim directly modifies the `site-packages` directory of the underlying Python installation. To ensure that these modifications are compatible with AWS Lambda, we embed λ-trim in the building phase of the container image. We use Amazon Linux Base as the base image. λ-trim deploys the resulting container image to AWS Lambda.

8 Evaluation

Our evaluation aims to answer these high-level questions:

- **(Q1) End-to-end latency, memory, and cost reduction:** Does debloating applications with λ-trim reduce the cold start latency, the memory footprint, and the total billed cost of applications in serverless platforms?
- **(Q2) Profiling effectiveness:** Does the profiling component of λ-trim select modules that are heavily affecting the application's performance?
- **(Q3) Debloating time:** How long does debloating take? Is λ-trim viable as a pre-deployment optimizer?
- **(Q4) Scaling:** Does λ-trim scale with K? What is the optimal K that keeps debloating time reasonable?
- **(Q5) Warm start performance:** Does λ-trim negatively affect warm start performance?
- **(Q6) Versus Checkpoint/Restore:** How does λ-trim compare to and complement C/R mechanisms?

Experimental setup. We perform the container build on Cloudlab's [16] `c6525-25g` machines that have Ubuntu 22.04, 16-core AMD 7302P 3GHz CPUs, and 128GB RAM before uploading and executing the final programs on AWS Lambda with the x86 ISA and Python runtime.

Benchmarks and methodology. We use the applications from Table 1 as our benchmarks. Unless otherwise noted, we use $K = 20$ and rank modules using their approximate marginal monetary cost. The oracle set for each application consists of 1–3 test cases. When the original benchmark (e.g., FaaSLight or RainbowCake) includes inputs, the set is taken from those benchmarks; otherwise, we manually generate examples to emulate simple, typical tasks that use the target library. Both the original and λ-trim-optimized applications are uploaded to AWS Lambda as Docker images.

We then perform 100 invocations and collect metrics from the AWS Lambda execution log. The input for each invocation comes from test cases in the oracle set. To trigger 100 cold starts, we update the function description field after each invocation request, forcing AWS Lambda to discard the warm function instance. For both cold and warm starts, we query the AWS log to ensure the invocation belongs to the desired start type and discard the data point otherwise.

8.1 (Q1) Latency, Memory and Cost Reduction

Figure 8 shows λ-trim's improvements to latency, memory footprint, and monetary cost.

End-to-end latency. End-to-end latency (E2E) measures the time between the user sending an invocation request and receiving a response from AWS Lambda. Several applications like **lightgbm**, **resnet**, **skimage**, and **spacy** show significant speedup. On average, λ-trim achieves 1.2× speed-up in E2E latency with a maximum of 2× for **resnet**.

There are several applications like **ffmpeg** and **image-resize** that do not benefit from λ-trim. These two applications use Python libraries that wrap the tools `ffmpeg` and `ImageMagick` and perform calls to their executables and are, therefore, bottlenecked on the corresponding system calls to these executables. In principle, these libraries could also be included in DD, but deployment would be more complex.

Memory footprint. Memory measures the runtime memory footprint of applications in MB. Multiple applications benefit heavily by using λ-trim, like **dna-visualization**, **lightgbm**, and **skimage**. These benefits come directly from removing redundant attributes from the module objects created from Python's import mechanism. Similarly to E2E latency, applications like **ffmpeg** and **image-resize** show little effect. On average, λ-trim achieves 10.3% improvement in memory with a max of 42% for **skimage**.

Monetary cost. Using Equation (1) and the actual memory footprint, applications like **dna-visualization**, **lightgbm**, **resnet**, **skimage**, and **spacy** all exhibit large improvements in cost. On average, λ-trim reduces cost by 19.7% with a max of 59% for **skimage**.

Since AWS Lambda has a minimum billing threshold for the configured memory (128 MB), applications requiring less

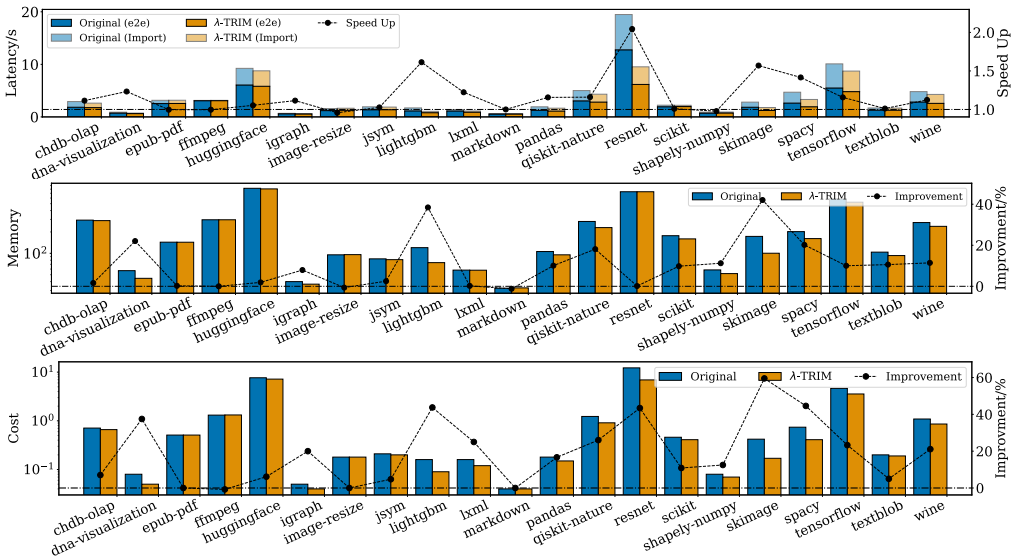

Figure 8. λ-TRIM's improvements to latency, memory footprint, and monetary cost for our benchmarked applications. The left axis and bars show the results for the original and trimmed versions. For latency, we show a breakdown of E2E versus Function Initialization time. The right axis and line graph show the relative improvement of λ-TRIM. The dashed line helps illustrate the speedup or improvement against the original application.

	Memory (MB)		Import Time (s)		E2E Latency (s)		
Application	FaaSLight	λ-TRIM	FaaSLight	λ-TRIM	Vulture	FaaSLight	λ-TRIM
huggingface	**-16.06%**	-2.11%	**−21.07%**	-10.21%	-2.30%	**−17.69%**	-6.65%
img-resize	**−3.23%**	-2.96%	**−7.77%**	-1.82%	-1.02%	**−11.10%**	-1.47%
lightgbm	−6.92%	**-38.44%**	−20.73%	**-54.81%**	-1.03%	−18.66%	**-30.50%**
lxml	**−3.23%**	-0.21%	−10.84%	**-41.58%**	-1.54%	−6.63%	**-19.37%**
scikit	−1.41%	**-9.8%**	−13.53%	**-19.60%**	-3.02%	**−12.83%**	-2.11%
skimage	**-42.98%**	-42.05%	**−69.27%**	-42.41%	-2.24%	**−42.05%**	-34.59%
tensorflow	−3.17%	**-9.01%**	−13.36%	**-15.58%**	-1.40%	−11.77%	**-15.50%**
wine	−6.09%	**-11.43%**	**−17.94%**	-13.73%	0.22%	**−14.72%**	-8.34%

Table 2. Comparison between reported improvements of FaasLight [30], Vulture [5] and λ-TRIM.

are billed as if they are using this minimum threshold, which hides λ-TRIM's memory benefit for small applications.

Comparison with FaaSLight and Vulture. We present a comparison with FaaSLight [30] and Vulture [5] in Table 2. We note that, similar to [47], we were unable to run the original tools to the same degree. Thus, we only compare against the reported numbers for their applications and metrics. We omit trivial use cases where all imports are unused.

Despite FaaSLight taking advantage of extensive manual annotations and intervention, the two systems show very similar performance in **skimage**, **tensorflow**, and **wine**. λ-TRIM seems to heavily outperform FaaSLight in **lightgbm** and **lxml**, while FaaSLight has greater improvements in **huggingface** and **image-resize**. Part of this difference may be due to smaller trial counts in FaaSLight's evaluation, as at 20 trials, our results still exhibited high variance. λ-TRIM has greater memory improvements in general, due to its more fine-grained handling of `from import` statements. Both systems outperform the reported [30] performance of Vulture.

8.2 (Q2) Ablation Study

Next, we conduct an ablation study to explore the effectiveness of various scoring methods for the λ-TRIM profiler. Specifically, we test 4 different scoring methods to rank the top K modules: (*a*) time, (*b*) memory, (*c*) combined, and (*d*) random. Time and memory methods rank modules based on the import time and the memory footprint, respectively, while the combined method utilizes Equation (2). Random randomly assigns values in the range $[0, 1]$ to each module.

The results from the ablation study are shown in Figure 9. We show results from a representative set of three applications. We can see that the combined scoring method constantly outperforms the other three methods, which showcases that the profiling phase, despite its approximations, correctly identifies modules with the largest impact on cost.

8.3 (Q3) Debloating Time and Efficacy

In Table 3, we present the total debloating time of each application, along with the most representative module's number of attributes before and after debloating. As mentioned in Section 5.3, we validate the output of each DD iteration by checking the standard output of the application. If we were to implement a call interceptor, there would be a small additional overhead in debloating time.

Debloating time ranges from minutes for small applications to 8 hours for the largest one (**huggingface**). The primary culprits are the ML libraries, e.g., `torch`, which consists of 3.9k files, and `transformers` with 1.9k files. We emphasize, however, that debloating time is off of the critical path—developers only apply λ-TRIM once, as the last step before deploying the application. There are also many techniques

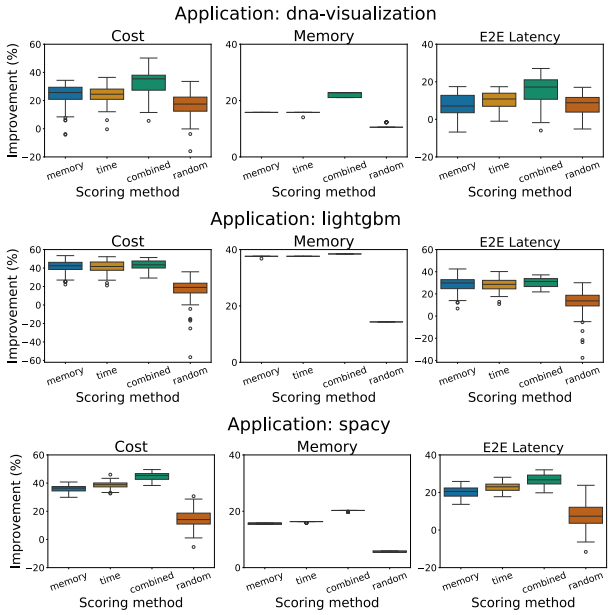

Figure 9. Cost, Memory and E2E improvement for different scoring methods.

that could be used to reduce this time. At a basic level, users can lower the number of modules to debloat (default is 20) to speed up the process. Prior work has also demonstrated the promise of learning techniques to choose the attribute set that is the most probable to pass the oracle test [25]. Finally, parallelization of DD may be possible; however, this is out of the scope of this paper as it likely requires novel approaches to dealing with dependencies between modules.

As for efficacy, λ-TRIM achieves a sizable reduction in attributes. For instance, λ-TRIM removes 3291 out of 3300 attributes from the `transformers` top-level module and 1306 out of 1414 attributes from `torch`. We also observe that the number of removed attributes for the same module varies between different applications. Specifically, it removes 496 out of 537 attributes from `numpy` for **dna-visualization**, while for **wine**, it only removes 33. This happens because different applications require different functionalities (and therefore different number of attributes) from the same module.

8.4 (Q4) Scalability and Optimal Debloating Size

We conduct experiments with varying numbers for K, i.e., the number of top modules to debloat. We again show only the results from 3 applications since most applications showcase the same behavior. The results are shown in Figure 10.

We observe improvements as the number of modules to debloat grows up until $K = 20$ from which point onwards there is a plateau in performance. This indicates that the modules that contribute the most during the import process have already been debloated and further debloating does not incur any performance benefits.

Application	Debloat Time (s)	Example Module	Attributes (Post/Pre)	Ckpt. Size (MB) (Post/Pre)
chdb-olap	44	chdb	11/32	39/41
dna-visualization	2142	numpy	496/537	14/17
epub-pdf	1878	pptx	20/38	36/37
ffmpeg	87	ffmpeg	35/46	11/11
huggingface	28756	transformers	3291/3300	240/255
igraph	159	igraph	137/185	11/13
image-resize	1973	wand.image	52/91	24/25
jsym	4385	sympy	914/938	37/41
lightgbm	4635	lightgbm	32/45	22/33
lxml	955	lxml.html	53/84	18/20
markdown	86	markdown	16/28	9/11
pandas	7066	pandas	125/141	36/41
qiskit-nature	1278	qiskit	30/49	224/244
resnet	26113	torch	1306/1414	80/84
scikit	4142	joblib	29/50	65/68
shapely-numpy	2393	shapely	161/176	15/17
skimage	3625	skimage	16/18	40/51
spacy	4722	spacy	36/60	85/99
tensorflow	10930	tensorflow	305/355	166/185
textblob	1561	nltk	550/560	25/29
wine	8573	numpy	33/537	87/95

Table 3. Benchmarked applications and λ-TRIM's effect on their debloating time ($K = 20$), C/R checkpoint size, and the attribute count of a representative module.

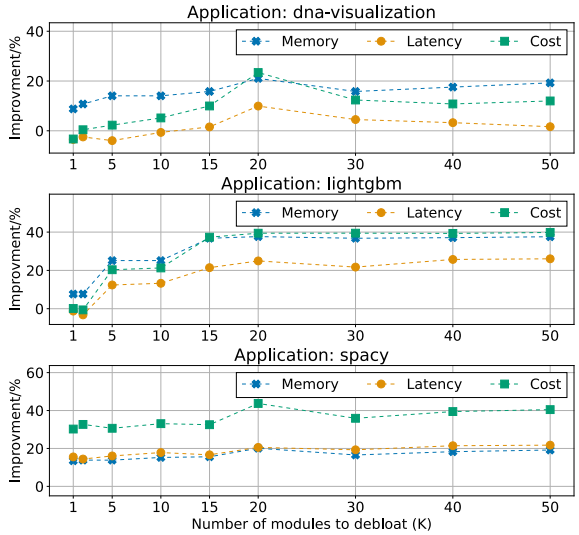

Figure 10. Varying K (number of modules to debloat).

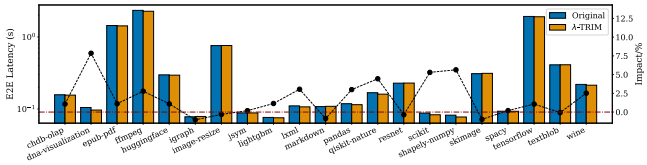

Figure 11. Warm start E2E latency impact of λ-TRIM.

Finally, memory and E2E latency seem to follow the same growth pattern. Cost also mimics the growth of these two factors, which is expected from Equation (1).

8.5 (Q5) Impact on Warm Starts

Figure 11 shows the difference in E2E latencies between the original and λ-TRIM applications during normal, warm-start invocation. The difference is less than 1 second, or 10%,

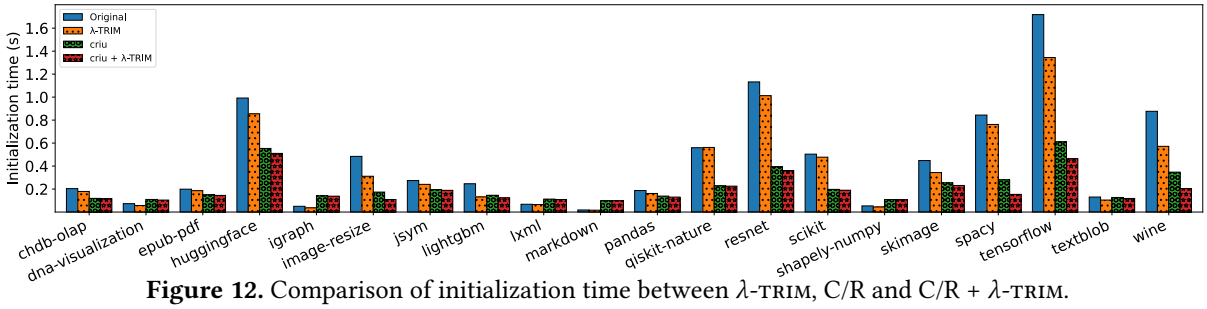

Figure 12. Comparison of initialization time between λ-TRIM, C/R and C/R + λ-TRIM.

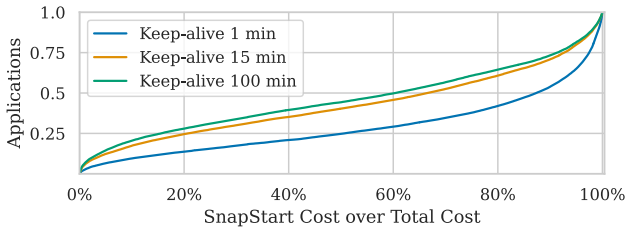

Figure 13. CDF of the ratio between SnapStart cost over total cost for functions in a simulated Azure trace [45]. Even with a keep-alive duration much longer than common practice, SnapStart doubles the cost of the majority of the applications.

Application		Original	λ-TRIM	Fallback Warm	Fallback Cold
dna-visualization	Cold	0.58	0.54	0.98	1.69
	Warm	0.10	0.09	0.14	0.61
lightgbm	Cold	0.98	0.80	1.09	2.06
	Warm	0.08	0.08	0.14	1.06
spacy	Cold	2.23	2.01	2.31	4.63
	Warm	0.08	0.08	0.14	2.31
huggingface	Cold	6.04	5.28	6.10	12.29
	Warm	0.31	0.29	0.35	6.31

Table 4. E2E latencies (in s) when triggering fallback. Original and λ-TRIM are the baseline E2E latencies with no error.

for all applications, which is expected as the behavior of a debloated application should stay the same as the original one. This small variation can be attributed external factors such as network fluctuations and AWS Lambda instance assignment, which we observed periodically through our extensive experimentation and are difficult to completely eliminate without large-scale longitudinal evaluations.

8.6 (Q6) Comparison with Checkpoint/Restore

Next, we compare the performance of λ-TRIM against C/R techniques, which also seek to reduce cold-start latencies.

C/R baselines. We evaluate against two strong baselines.

The first is C/R prototype based on `CRIU` [39], which is the state-of-the-art C/R tool in userspace. `CRIU` can freeze a running application and checkpoint it to disk so that it can be restored later from the point at which it was frozen. In the case of cold starts, the checkpoint should be taken right after the initialization but before the handler. When another cold start is triggered, `CRIU` can restore the state of the function from the checkpoint. Note that CRIU requires either the `CAP_CHECKPOINT_RESTORE` Linux capability, which cannot be set in AWS Lambda. Results in this section are instead based on a Docker container on a local machine with Ubuntu 24.04, 16-core Intel Ultra 7 155H, and 16 GB RAM.

The second baseline is AWS SnapStart [3], an optional feature that takes an encrypted, VM-level snapshot of serverless functions. While SnapStart is a production feature, it is currently limited to very small function sizes, preventing us from evaluating our baselines directly. Rather, our results here are mainly with the aid of simulation.

In both cases, we compare the original application against C/R, λ-TRIM, and the combination of the two.

Initialization time versus `CRIU`. Figure 12 compares the initialization time of all evaluated variants. We observe significant differences between applications, largely based on their initialization time.

For small applications (<0.2 s), λ-TRIM outperforms all other variants. In fact, C/R is much worse than the baseline application. This is due to the fact that `CRIU` recreates the process tree by forking its own process and then restores process state by using information collected from reading `/proc` during the checkpointing. This procedure incurs an overhead, which seems to be around 0.1 seconds.

For larger applications, pure C/R begins to outperform pure λ-TRIM. An exception is **lightgbm**, which benefits significantly from debloating. C/R becomes more effective as we look at larger applications since loading memory pages from the checkpoint image is much faster than file I/O and command execution by the Python interpreter. In addition, the initialization phase includes not only library imports but also environment/model loading, an action that λ-TRIM cannot optimize. This is the case in **spacy**, which needs to load a language model to perform a simple NLP task.

The two techniques are, however, complementary as λ-TRIM can be used to reduce the size of the checkpoint image. Table 3 shows the checkpoint size produced by `CRIU` and by `CRIU`+λ-TRIM. Debloating always reduces the size of the checkpoint and does so by an average of 11%.

Monetary costs of using SnapStart. The tradeoff of C/R-based approaches are its large resource overheads to store

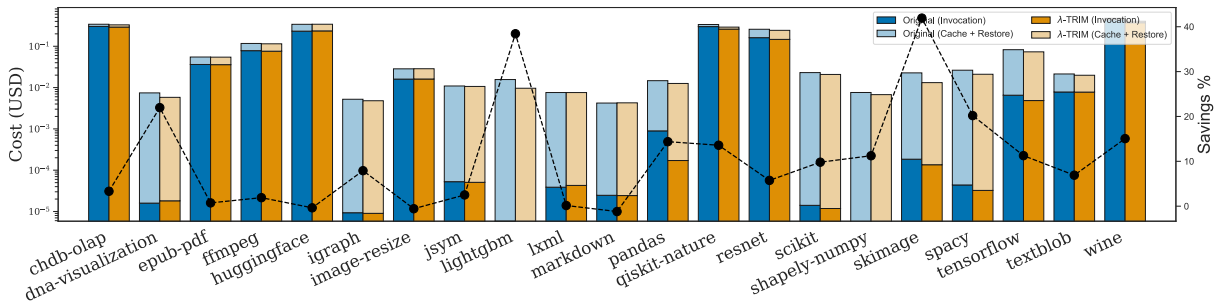

Figure 14. Amortized invocation and SnapStart costs for simulated traces of our benchmarked applications. Simulated based on an Azure trace [45] and AWS SnapStart pricing [4], assuming a 15-minute keep-alive time.

and restore the checkpoints. We can quantify these overheads using the pricing of SnapStart, which charges users based on both the restore cost (number of cold starts) and storage costs (quantifed in units of GB-seconds) [4].

To illustrate the magnitude of these costs, we simulate running applications in the Microsoft's Azure Function trace [45] with SnapStart. Figure 13 shows a CDF of the ratio between SnapStart costs and the total cost for the applications. Even for extremely long keep-alive times, the median application would spend >60% of its cloud budget on paying for C/R support, mostly on caching costs.

To estimate λ-TRIM's potential effect on these costs, we take each of the applications in Table 1 and find the most similar function in the entirety of the Azure trace. Similarity is quantified as the L2 norm of memory and duration. We then simulate the benchmarked application over 24 hours using the associated function's invocation traces (assuming functions stay warm for at least 15 mins). As shown in Figure 14, λ-TRIM reduces total costs by up to 42% (average of 11%) by reducing the memory footprint and checkpoint size.

8.7 Fallback Overhead

λ-TRIM can fall back to the original function if necessary attributes are incorrectly removed. When triggering the fallback, the overheads include setup, communication delays, and invocation of the original function. We comprehensively evaluate these overheads by measuring E2E latencies in every combination of warm/cold start for the λ-TRIM and the fallback functions. Table 4 shows results for representative applications of different sizes: small (**dna-visualization**), medium (**lightgbm**), and large (**spacy, huggingface**).

The setup overhead is around 50 ms, measured by timestamps in the function. When the fallback function is cold, its cold start latency dominates the fallback overhead. Cold fallback overhead doubles the E2E latency of a cold λ-TRIM function and contributes over 90% of the latency of a warm λ-TRIM function. Overall, the invocation of the original function is the main source of fallback overhead.

9 Related and Future Work in Debloating

Expanding Section 3.1, λ-TRIM is also related to the extensive work in debloating such applications with techniques like static and reachability analysis [5, 7, 37], dynamic analysis [24, 37], just-in-time loading [33] or even manual investigation and modification of applications [12, 47]. Like λ-TRIM, these systems are motivated by the fact that modern software is heavily bloated due to the use (and reuse) of libraries offering a plethora of functionalities [19, 27, 36]. λ-TRIM is based on similar techniques to conventional debloaters, but is the first to specifically target serverless applications and their unique structure, execution model, and optimality criteria.

Under the umbrella of debloating, DD is a prominent technique, but it has been constrained to statically typed languages like C/C++ [53] or, very recently, dynamically typed compiled languages [40]. This technique and efforts to improve it (e.g., using learning to accelerate the search for the reduced program [25]) are complementary to λ-TRIM.

Looking forward, although developers pay the cost of debloating once (and therefore, this cost is off the critical path), λ-TRIM still suffers from substantial debloating times for medium to large applications. We plan on accelerating the debloating phase with various optimizations.

First, we will parallelize DD both intra-(multiple sets of attributes of the same module in parallel) and inter-(multiple modules in parallel) modules. The latter will require very meticulous handling of module dependencies, mainly due to Python's cyclic imports. Finally, we plan to implement a continuous debloating pipeline for both function updates and inputs that are collected through our fallback mechanism. This pipeline will make use of logs collected during the initial debloating to drive the subsequent debloating more efficiently in both aforementioned cases.

10 Conclusion

This paper introduced λ-TRIM, a system designed to reduce the overhead of Python-based serverless applications by optimizing their Function Initialization phase. This phase has an outsized effect on not only cold start latency but also the resource consumption and monetary costs of all executions—cold or warm. λ-TRIM leverages profiling and DD, and offers a practical and immediately deployable solution that aligns well with other cold start optimization efforts while contributing uniquely to cost efficiency.

Acknowledgments

We gratefully acknowledge our shepherd, Pedro Fonseca, and the anonymous ASPLOS reviewers for all of their thoughtful comments. This work was funded in part by NSF grants CNS-2107147, CCF-2326606, and CNS-2321726.

References

[1] [n. d.]. GitHub – WenJinfeng/FaaSLight. https://github.com/WenJinfeng/FaaSLight. [Accessed 09-14-2024].

[2] 2019. Provisioned Concurrency for Lambda Functions. https://aws.amazon.com/cn/blogs/aws/new-provisioned-concurrency-for-lambda-functions/. [Accessed 18-10-2024].

[3] 2025. Improving startup performance with Lambda SnapStart - AWS Lambda — docs.aws.amazon.com. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html. [Accessed 09-03-2025].

[4] 2025. Serverless Computing – AWS Lambda Pricing – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/lambda/pricing/#SnapStart_Pricing. [Accessed 09-03-2025].

[5] 2025. Vulture: Find dead Python code. https://github.com/jendrikseipp/vulture. [Accessed 10-03-2025].

[6] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[7] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 70–83.

[8] Amazon Web Services 2024. Documentation – AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/python-handler. [Accessed 17-09-2024].

[9] Amazon Web Services 2024. Profiling functions with AWS Lambda Power Tuning - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/operatorguide/profile-functions.html. [Accessed 17-09-2024].

[10] Amazon Web Services 2024. Serverless Computing – AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/. [Accessed 17-09-2024].

[11] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS made fast using snapshot-based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 730–746. https://doi.org/10.1145/3492321.3524270

[12] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1697–1714. https://www.usenix.org/conference/usenixsecurity19/presentation/azad

[13] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3472883.3486992

[14] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/atc23/presentation/brooker

[15] Chen Chen, Lars Nagel, Lin Cui, and Fung Po Tso. 2023. S-Cache: Function Caching for Serverless Edge Computing. In *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking* (Rome, Italy) *(EdgeSys '23)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3578354.3592865

[16] Cloudlab [n. d.]. CloudLab - A testbed for cloud computing research. https://www.cloudlab.us/.

[17] Datadog. [n. d.]. The State of Serverless 2020 — datadoghq.com. https://www.datadoghq.com/state-of-serverless-2020/. [Accessed 09-11-2024].

[18] Datadog. 2025. The State of Serverless. https://www.datadoghq.com/state-of-serverless/. [Accessed 10-03-2025].

[19] Georgios-Petros Drosos, Thodoris Sotiropoulos, Diomidis Spinellis, and Dimitris Mitropoulos. 2024. Bloat beneath Python's Scales: A Fine-Grained Inter-Project Dependency Analysis. *Proc. ACM Softw. Eng.* 1, FSE, Article 114 (July 2024), 24 pages. https://doi.org/10.1145/3660821

[20] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[21] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[22] Python Software Foundation. 2003. PEP 302 – New Import Hooks. https://peps.python.org/pep-0302/#specification-part-1-the-importer-protocol. https://peps.python.org/pep-0302/#specification-part-1-the-importer-protocol

[23] Alexander Fuerst and Prateek Sharma. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[24] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 248–258. https://doi.org/10.1145/2610384.2610407

[25] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[26] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[27] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1, Article 03 (May 2020), 27 pages.

[28] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/atc22/presentation/li-jie

[29] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[30] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. FaaSLight: General Application-level Cold-start Latency Optimization for Function-as-a-Service in Serverless Computing. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 119 (July 2023), 29 pages. https://doi.org/10.1145/3585007

[31] A Cloud Guru News. [n. d.]. How long does AWS Lambda keep your idle functions around before a cold start? https://www.pluralsight.com/resources/blog/cloud/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start. [Accessed 09-14-2024].

[32] Shanxing Pan, Hongyu Zhao, Zinuo Cai, Dongmei Li, Ruhui Ma, and Haibing Guan. 2024. Sustainable Serverless Computing With Cold-Start Optimization and Automatic Workflow Resource Scheduling. *IEEE Transactions on Sustainable Computing* 9, 3 (2024), 329–340. https://doi.org/10.1109/TSUSC.2023.3311197

[33] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 164–180. https://doi.org/10.1145/3385412.3386017

[34] PyPI [n. d.]. PyPI - The Python Package Index. https://pypi.org/. [Accessed 17-09-2024].

[35] PyPI 2024. Statistics · PyPI. https://pypi.org/stats/. [Accessed 17-09-2024].

[36] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (Dallas, Texas, USA) *(FEAST '17)*. Association for Computing Machinery, New York, NY, USA, 65–70. https://doi.org/10.1145/3141235.3141242

[37] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[38] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 753–767. https://doi.org/10.1145/3503222.3507750

[39] Jesse Ruderman. 2012. CRIU - Checkpoint/Restore In Userspace. https://criu.org/.

[40] Jesse Ruderman. 2016. Lithium. https://github.com/MozillaSecurity/lithium.

[41] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[42] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 714–729. https://doi.org/10.1145/3492321.3524272

[43] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2024. FUSIONIZE++: Improving Serverless Application Performance Using Dynamic Task Inlining and Infrastructure Optimization . *IEEE Transactions on Cloud Computing* 12, 04 (Oct. 2024), 1172–1185. https://doi.org/10.1109/TCC.2024.3451108

[44] Biswajeet Sethi, Sourav Kanti Addya, and Soumya K. Ghosh. 2023. LCS: Alleviating Total Cold Start Latency in Serverless Applications with LRU Warm Container Approach. In *Proceedings of the 24th International Conference on Distributed Computing and Networking* (Kharagpur, India) *(ICDCN '23)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/3571306.3571404

[45] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 14, 14 pages.

[46] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3423211.3425682

[47] Syed Salauddin Mohammad Tariq, Ali Al Zein, Soumya Sripad Vaidya, Arati Khanolkar, and Probir Roy. 2024. LibProf: A Python Profiler for Improving Cold Start Performance in Serverless Applications. arXiv:2406.11734 [cs.SE] https://arxiv.org/abs/2406.11734

[48] Kanchan Tirkey, Anisha Kumari, Sagarika Mohanty, and Prof. Bibhudatta Sahoo. 2023. A Novel Function Fusion Approach for Serverless Cold Start. In *2023 International Conference on Communication, Circuits, and Systems (IC3S)*. 1–5. https://doi.org/10.1109/IC3S57698.2023.10169477

[49] Luc van Donkersgoed. [n. d.]. When is the Lambda Init Phase Free, and when is it Billed? https://lucvandonkersgoed.com/2022/04/09/when-is-the-lambda-init-phase-free-and-when-is-it-billed/. [Accessed 09-14-2024].

[50] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 416–428. https://doi.org/10.1145/3468264.3468558

[51] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 335–350. https://doi.org/10.1145/3617232.3624871

[52] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 253–267.

[53] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (feb 2002), 183–200.

https://doi.org/10.1109/32.988498

[54] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[55] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 653–669. https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

A Artifact Appendix

A.1 Abstract

λ-trim is a debloater for Python applications. Given a Python function and a set of inputs to this function, λ-trim automatically removes all redundant modules, functions, and classes from the modules that the application imports. This artifact contains instructions to install λ-trim and scripts to reproduce key results of our paper (Figures 8, 9, 10, 11, 12, 13, 14).

The source code of λ-trim is available at https://github.com/eniac/lambda-trim and scripts for all experiments are available at https://github.com/xutingl/lambda-trim-artifact.

A.2 Artifact check-list (meta-information)

- **Algorithm: Cost-driven Delta Debugging for Debloating**
- **Run-time environment: Python 3.10**
- **Metrics: E2E latency, memory, billed duration, and import time.**
- **Output: λ-trim outputs optimized serverless application.**
- **Experiments: Figures 8, 9, 10, 11, 12, 13, 14.**
- **How much disk space required (approximately)?: 200GB if keeping Docker images for all appliccations. 50GB if cleaning up images after each experiment.**
- **How much time is needed to prepare workflow (approximately)?: 2-3 hours.**
- **How much time is needed to complete experiments (approximately)?: 3-4 days if experiments are run serially.**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: GPL-3.0**

A.3 Description

A.3.1 How to access

https://github.com/xutingl/lambda-trim-artifact

A.3.2 Hardware dependencies

λ-trim is designed for serverless applications and does not require special hardware.

A.3.3 Software dependencies

Requires Docker and AWS CLI. The complete list of dependencies is provided in the repository README.

A.4 Installation

λ-trim can be installed with

```
$ cd lambda-trim
$ pip install -e .
```

A.5 Experiment workflow

Complete instructions and explanations of the experiment workflow are included in the repository README. We provide an overview below.

A.5.1 Debloating (Figure 8)

Run the following to create and run baseline functions and λ-trim debloated functions.

```
$ python experiments/debloating.py --action create-baseline
$ python experiments/debloating.py --action run-baseline
$ python experiments/debloating.py --action create-debloat
$ python experiments/debloating.py --action run-debloat
```

λ-trim runs in the `create-debloat` step. It may take 30 minutes (jsym) to 8 hours (huggingface) to debloat an application.

Use `experiments/debloat/fig8.ipynb` to generate Figure 8.

A.5.2 Ranking (Figure 9)

To run the experiments for the various scoring methods (memory, time, combined, random), run the following:

```
$ ./experiments/ablation/run_all.sh ranking
```

If you want to run a specific application `appname` for the various scoring methods, you can run:

```
$ ./experiments/ablation/run_scoring.sh <appname>
```

Use `experiments/ablation/plot_scoring.ipynb` to generate Figure 9. This step assumes that you first run the debloating experiment (Figure 8).

A.5.3 Varying K (Figure 10)

To run the experiments for varying K (number of modules to debloat), run the following:

```
$ ./experiments/ablation/run_all.sh k
```

If you want to run a specific application `appname` for varying K, you can run:

```
$ ./experiments/ablation/run_k.sh <appname>
```

Use `experiments/ablation/plot_varying_k.ipynb` to generate Figure 10. This step assumes that you first run the debloating experiment (Figure 8).

A.5.4 Warm Starts (Figure 11)

Warm-start experiments use the same functions created in the debloating experiment (Figure 8). This step can be **skipped** if baseline and debloated Lambda functions have been created in the debloating experiment (steps 1 and 3 in

the debloating experiment). Otherwise, you need to create them by running

```
$ python experiments/debloating.py --action create-baseline
$ python experiments/debloating.py --action create-debloat
```

Then, run warm-starts for baseline and debloated functions

```
$ python experiments/debloating.py --action run-baseline-
    warm
$ python experiments/debloating.py --action run-debloated-
    warm
```

Use `experiments/warm/fig11.ipynb` to generate Figure 11.

A.5.5 Comparison with Checkpoint/Restore (Figure 12)

For our comparison with Checkpoint/Restore (CR) techniques (Figure 12), we built a prototype with CRIU. The prototype spawns a CRIU server, and the application connects to the server through a gRPC call to force a self-dump/checkpoint. Afterwards, we invoke the application by issuing a restore call to the CRIU server.

We are testing four variants:

- Original application
- Original application with CR
- Debloated application
- Debloated application with CR

To speed up the building process, we provide a base Docker image (spyrospav/criu-debloat:latest) that contains a minimum CRIU build.

For a single application app, you can reproduce the comparison by running:

```
$ ./experiments/cr/run.sh app
```

Note that this creates a Docker container for each variant and executes the test.

To build all the applications, run

```
$ ./experiments/cr/run_all.sh
```

Use `experiments/cr/analyze_cr.ipynb` to interactively produce the bar plots with the results for both a single application and the whole benchmark set after running the experiments (Figure 12).

A.5.6 Checkpoint size (Table 3 – Ckpt. Size column)

The size of the checkpoints (Table 3 – Ckpt. Size column) for both the original and the debloated application is saved in the directory `experiments/cr/output/` after running the CR experiment.

A.5.7 Fallback (Table 4)

Create undebloated Lambda functions to be used as fallback functions. This step can be skipped if the baseline functions **dna-visualization**, **lightgbm**, **spacy**, and **huggingface** have been created in step 1 of the debloating experiment (Figure 8). Otherwise, create them by running

```
$ python experiments/debloating.py --action create-baseline
    --single-app dna-visualization
$ python experiments/debloating.py --action create-baseline
    --single-app lightgbm
$ python experiments/debloating.py --action create-baseline
    --single-app spacy
$ python experiments/debloating.py --action create-baseline
    --single-app huggingface
```

Then, run fallback experiments with

```
$ ./experiments/fallback/run_fallback.sh
```

A.5.8 SnapStart Simulation (Figures 13 and 14)

Use `experiments/snapstart/fig13_14.ipynb` to run simulation experiments and produce Figures 13 and 14. The simulation is based on traces from the Azure functions dataset [45], which will be downloaded in the notebook.

A.6 Evaluation and expected results

Serverless applications are invoked by default 100 times, and results will be stored in `results` directory for each experiment. Due to the nature of serverless cloud services (like user traffic, network, etc.), the exact numbers may differ from those in the paper. We provide results to generate figures in `paper_results`.