Profile-Guided Optimization of Cold Starts in Serverless Applications with ColdSpy

by

Ali Al Zein

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer and Information Science)
in the University of Michigan - Dearborn
2024

Master's Thesis Committee:

Assistant Professor Probir Roy, Chair
Associate Professor Jinhua Guo
Assistant Professor Birhanu Eshete
Assistant Professor Foyzul Hassan

Ali Al Zein

alielzei@umich.edu

ORCID iD: 0009-0009-0770-5680

Dedication

To Mama, Baba, Fufu, Nano, Sama, Ibrahim, and most importantly, Yaso.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

ABSTRACT

Serverless computing offers significant benefits over past traditional execution models, promising efficient resource utilization, cost-effectiveness, and extreme elasticity. Despite its benefits, serverless applications face challenges due to the ephemeral nature of serverless functions, leading to "cold-start" latencies. Prior research has addressed cold-start latencies by designing novel container techniques such as container caching, sharing, and memory optimizations. However, none of the research has explored a measurement-based approach to identify code-level inefficiencies that cause significant cold-start latencies. In this research, we first investigate serverless applications to identify the common inefficient library initialization and usage patterns that result in significant cold-start latency. We further generalize the patterns and propose a novel dynamic program analysis tool, $ColdSpy$, to detect the inefficiency and guide the developers for optimization. Guided by $ColdSpy$, we optimize five real-world serverless applications resulting in the reduction of cold starts up to 42% in AWS Lambda.

CHAPTER 1

Introduction

The evolution of cloud computing has led to the emergence and growing popularity of the serverless architectural paradigm. The serverless architecture abstracts the server management away from the developers, allowing them to build and run serverless applications without the burden of manging the infrastructure. Serverless further allows to scaling the resources according to the needs of the applications. Due to the fine-grained control over resources, serverless enables better utilization of the cloud resources among the tenants. Finally, serverless enables a better pricing model where the users only pay for the resource the function uses.

Despite its advantages, serverless computing has its challenges. A serverless function is ephemeral in nature as its resources are reclaimed for other workloads once it stops being invoked after idling for a set Time-To-Live (TTL) duration in which it is said that the function is warm. When it is invoked again, it is re-initialized, taking a duration that is often referred to as *cold-start*. As function invocations are usually distributed in no predictable pattern, a function has to re-initialize repeatedly impacting response latency.

This latency is further aggravated by the heavy runtime of the programming languages it supports such as Python and JavaScript, particularly during library initialization that constitutes a major part of cold-start time, as at every cold-start libraries will need to be reloaded.

Several approaches have been proposed to address these runtime inefficiencies. Previous research performs optimizations on the platform, deployment, and code level. This research focuses on addressing inefficiencies during the library initialization stage of the serverless application runtime. We introduce $ColdSpy$, a dynamic program profiling tool designed for serverless applications that

detects four inefficient programming patterns.

$ColdSpy$ measures two main aspects of a serverless Python application runtime; 1) the time it takes to import a module and 2) its utilization rate. $ColdSpy$ ranks imported modules based on an inefficiency score, giving developers a priority list of the modules to analyze for optimization. $ColdSpy$ also builds a calling context tree by collecting the calling context at every sample, allowing developers to further trace module utilization through profiling as well as aid in debugging. Using specialized software, data generated by $ColdSpy$ is then easily parsed and visualized to be browsed. Developers can take advantage of the insights provided by $ColdSpy$ to analyze their serverless applications, uncovering optimization opportunities in the application code that brings cold-start time down.

Previous work explored static analysis to follow the call paths by inspecting the code and its call hierarchy in isolation without a runtime, transforming it by compressing parts of the code to reduce loading times and hence cold-start time. However, static analysis brings challenges in accurate detection by lacking a measurement approach that can prioritize inefficiencies. For each of the four inefficient programming patterns that are discussed, $ColdSpy$ detects different optimization strategies. The effectiveness of the tool is evaluated by analyzing real-world applications, applying optimizations, and measuring the improvements.

To summarize, this work aims to contribute to the ongoing evolution of cloud computing by exploring the inefficiencies of library loading particularly in heavy programming language runtimes in a serverless environment. The contribution and structure of this thesis are as follows: Chapter 2 discusses the background and related work to serverless computing, the cold-start problem, and the heavy runtime of Python. Chapter 3 introduces four inefficiency patterns commonly found in serverless applications. Chapter 4 describes the design and implementation of $ColdSpy$. Finally, in Chapter 5, $ColdSpy$ is evaluated by performing case studies on three real-world applications after which we conclude in Chapter 6.

CHAPTER 2

Background and Related Work

2.1 Serverless computing

One traditional alternative to serverless computing is server-based computing. Developers used this paradigm to host their applications on physical, virtual, or cloud servers. This has long been to make applications available to the public. Yet, as this method has long posed challenges of optimal resource utilization, serverless has been introduced.

Serverless is a cloud computing execution model where the cloud provider manages the deployment of application code and the allocation of machine resources. Developers only have to upload their code. Then, through drop-down lists, they can choose options like memory size, runtime, and CPU architecture for the application to run on.

When the application is invoked, the cloud provider provisions a sand-boxed container with resource access such as ephemeral storage, specified memory, and processing power, then the code is loaded to make the application ready for execution. However, during no traffic, no resources are instantiated or consumed. After the serverless application instance finishes execution and returns a response, it will idle for a Time-to-Live duration before the resources get reclaimed for other uses by the cloud provider. While this process is abstracted away from the perspective of a developer, its temporary behavior is key to the advantages that come with serverless.

The serverless architecture greatly minimizes the developer's concern of managing the underlying infrastructure. Serverless abstracts the burden of instantiating a virtual machine in the cloud while achieving efficient usage. Developers who choose to deploy their applications on serverless

need not worry about many of the challenges posed by server-based applications. The cloud provider automatically manages most of the serverless instance's configuration. Serverless is fault tolerant; new execution instances will just be re-created if older ones crash. Developers do not need to plan the capacity for their application, deeply configuring or maintaining its infrastructure. Finally, cloud providers automatically scale the number of instances enabling the processing of any amount of oncoming load concurrently.

Serverless function customers are only charged for the time instances compute for. They are not charged for the time when the function had allocated resources but did not serve any invocations. Serverless billing is often cheaper. This is especially true for applications with variable or burst loads. Customers only pay for the time during high traffic. They would incur no extra charges when usage is low or when a server would have been idle.

Serverless removes the disadvantages and burden of server-based computing. It allows developers to deploy their applications to the cloud while saving time. This faster development process boosts productivity and lowers the entry barrier for new developers. This reduces the time for organizations to get their applications to market. It also cuts billing and management costs.

2.2 The Cold-Start Problem

The serverless model promises scalability, cost efficiency, and less complexity. But, it also brings a unique challenge: the cold start problem. Upon invocation, it is possible that the function is not in a state where its resources are already claimed, a state often referred to as *warm*. However, starting up the runtime, loading the code, and loading libraries all add to the time before the code runs. This is often referred to as the *cold-start* time.

Cold-start time occurs only when there are no existing warm instances to process an invocation. This happens when there is a sudden increase in traffic. In serverless settings, applications have variable load. They are invoked in unpredictable patterns. This reduces the effectiveness of pre-warming the function with predictive deployment strategies. This increases the likelihood of encountering a cold start which consequently causes higher response delays.

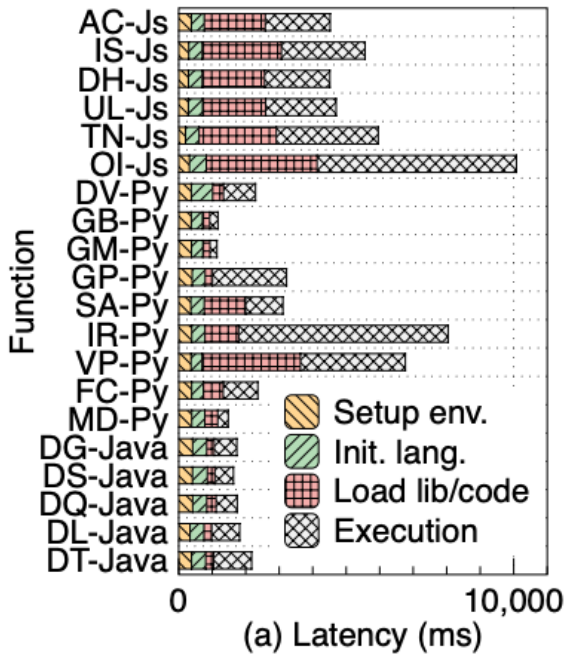


Figure 2.1: Cold-start latency footprint breakdown of three stages for 20 realistic functions of Node.js, Python, and Java [11]

The cold-start problem is a disadvantage that has been studied since the serverless architecture has been introduced. Significant research went into optimizing cold-start times on multiple levels. On the platform level, container initialization overhead is reduced through advanced techniques such as caching, checkpointing, [10] memory deduplication, [23] and container sharing [21]. Additionally, cold-start time is reduced using deployment strategies that perform predictive instance initialization ahead of time such as keep-alive [8] and pre-warming [14]. In contrast, on the code level, little contribution has been made to serverless optimization.

Optimizing on the platform level is agnostic of the application's implementation-specific code. However, inefficient code can very possibly cause longer cold-start time delays. In serverless, the code is often just a runner script that makes use of libraries, and the time it takes to load libraries constitutes a part of cold-start time. Therefore, inefficient code practices will increase library loading times and also drive cold-start time up.

Previous work explored static analysis in Python to follow the call paths of the code base and

its call hierarchy in isolation without a runtime. Transforming the code by compressing parts of it proved to reduce loading times and hence mitigate cold-start [17]. However, static analysis brings challenges in accurate detection by lacking a measurement approach that can prioritize inefficiencies.

There are many performance analysis tools available for use in Python. Scalene [7], PySpy [13] and cProfile [12] among others are used by developers to collect information about which call paths are visited and where the code spends its time. These tools however are not developed with serverless in mind. They do not measure the import time of a module or calculate its rate of utilization.

2.3 Python

Python is a popular serverless programming language. It is one of the prominent options, known for its simplicity, versatility, and wide adoption. It is a compelling option for developers to choose as it has a straightforward syntax, a comprehensive standard library, and a robust community of third-party packages. It facilitates rapid development and deployment of applications, making it a popular and versatile tool for a variety of domains—from web development to data analysis and machine learning.

Python's ease of use makes it a time-saving programming language enabling quick iteration and deployment of services. This makes it a suitable pairing of serverless as they are both overlapping on the feature of providing a faster development experience.

Despite being an advantageous language in a serverless environment in terms of fast development, Python is an interpreted language and is characterized by its performance limitations due to its heavy runtime, starting a Python program can take more than 10 times the startup time of an equivalent C++ program [2]. The lack of pre-compilation in Python results in the code being parsed and interpreted each time it is executed. In high-performance computing scenarios or applications where low response latency is critical, this characteristic of Python can be a significant drawback.

Although Python's extensive third-party packages available to developers are a big part of its

flexibility and strength, the process of importing these packages and the modules they contain cause a significant startup time delay.

When a Python script imports a module, the interpreter finds it, compiles it into bytecode, and then loads it into memory. This abstraction could be simple to understand by a developer allowing for easy code reuse and modular programming. However, the import system design choices and Python's slow runtime increase the startup time significantly. Before executing the code, a serverless function imports all modules it depends on. Every time a serverless function is executed from a cold state the duration of the import time will have to be spent again causing more response delays

The Python import system causes the serverless function to execute all the import statements before ever executing the business logic it contains. It is possible, however for the import to not be called during the entirety of its life, from the time its resources are reserved until they are freed. In other cases, a module could have been imported for unnecessary work that can be avoided in the processing of the function. In addition, a module could have equivalent implementations that take less time to import. These inefficiencies demonstrate the possibility of avoiding some imports saving cold-start time and they constitute inefficiencies that hint to the possibility of being detectable.

Future improvements in Python's performance, particularly in reducing module load times and optimizing runtime efficiency, are crucial for enhancing its suitability in serverless applications where low latency and efficient resource utilization are paramount. This aligns with ongoing research and development efforts aimed at mitigating the cold-start impact, ensuring Python continues to evolve as a robust solution in the serverless domain.

CHAPTER 3

Inefficiency Pattern Categories

After manually investigating multiple of serverless applications, we identify common code inefficiency patterns. In this section, four categories are discussed that represent code inefficiencies.

3.1 Never used

In the first pattern, we consider a scenario in which the application code includes a helper function that depends on a third-party library. In addition, only that function throughout the whole code needs that library. If the function itself is never called, loading the library becomes unnecessary.

By identifying and eliminating such unused libraries, the application start-up time can be reduced, which is critical in reducing a cold-start delay. The developer is required to confirm that the function is indeed unused by inspecting the code, and then the developer removes or comments out the function and the library loading.

```
1  import pathlib2
2  ...
3
4  def path_matches_any(text, patterns):
5      """Check if the text matches any of the given wildcard patterns
6      using the pathlib2 library
7      ...
8      """
9      if not isinstance(text, str):
10         return False
```

```
11      return any(pathlib2.PurePath(text).match(pattern) for pattern in patterns)
```

Listing 3.1: Python function demonstrating an unused import

In Listing 3.1, a library module, *pathlib2*, is imported and used in a function. However, this function is not utilized anywhere in the application.

In cases like these, developers can significantly reduce cold-start times simply by removing or commenting out the helper function and unnecessary module imports.

3.2 Rarely used

In this section, we explore the inefficiency created when an application loads a library but rarely uses it. Such a scenario occurs when a library is loaded and referenced. However, calling that library is rare, often due to the nature of the input data and the existence of conditional logic.

```
1 import HeavyLib
2 if len(payload) > 1000:
3     processed_payload = HeavyLib.process_data(payload)
4     return processed_payload
5 else:
6     return payload
```

Listing 3.2: Code snippet demonstrating conditional import using a mock example

As illustrated in Listing 3.2, the *HeavyLib* library is imported but only used when certain conditions are met based on the input data. Specifically, when the size of the payload exceeds a certain threshold. This exemplifies a situation where a library import, which might incur a lengthy load time, is rarely utilized.

In some cases, input data reaches the serverless application without resulting in enough calls to a library to justify the cost of its long loading time. This inefficiency category touches on serverless applications that load a library to cover possible cases it might be needed. The library might be called once or twice, or maybe never, in the application's dynamic runtime. The developer who adds the library to their code does not consider how often the module is called.

When the developer is encountered with this inefficiency pattern they can trace where the library was called from the code and load it just in time before the library call. As the library is not often called, the optimization would speed up function instances that don't import the module when they start including the ones that never utilize the library.

By identifying and optimizing the conditional loading of libraries that are infrequently used, as shown in Listing 3.2, there is an opportunity to optimize application performance. This approach helps in alleviating the cold-start latency by aligning resource utilization more closely with the dynamic nature of serverless applications.

3.3 Feature redundancy

Workloads hosted on serverless are designed with the primary goal of applying computational operations on given data and outputting the result. Certain libraries come with additional features such as debug logs, progress bars, or interactive interfaces that could be appreciated in local user development or server-based environments but may be irrelevant in a serverless setting.

In this section, we see how OCRmyPDF, a popular Python library provides a great example of this inefficiency pattern. It requires the Rich library which is used to display progress bar information and other logging tools related to PDF processing. These are ultimately logged to CloudWatch but not even used to their full extent.

As CloudWatch does not support features like text color as illustrated in Figure 3.1. The features of the Rich library are often not properly leveraged in a serverless environment. Not only do batch workloads or users invoking the serverless application not need to see log entries but, normally, they do not even have access to them. Rich is a heavyweight library that was observed to be taking significant application initialization time and so the feature is removed along with references to the console library.

The discussion in this section further highlights how inefficiencies can arise from inefficient programming practices by including features that do not align with the computational goal of a serverless function. By carefully considering real-world usage, developers can significantly optimize

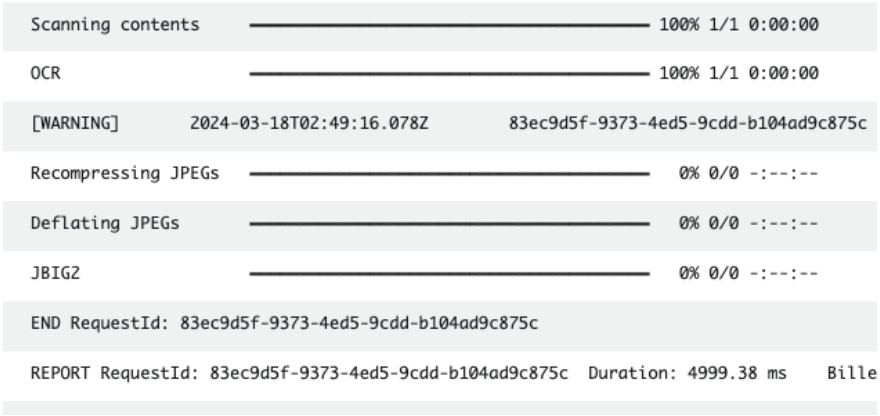


Figure 3.1: OCRmyPDF logging the progress of the PDF processing to CloudWatch using the Rich library

the performance and cost-efficiency of their serverless applications.

3.4 Library choice

When writing code, software developers often rely on pre-developed libraries that provide robust and maintained code. Instead of implementing functionalities from scratch, developers have a choice of multiple libraries available to achieve identical functionalities.

Although two libraries could offer identical functionality, it is likely that their impact on application performance is different, particularly during initialization time. For example, a code that loads a comprehensive library to only use a single function it provides could be suffer from long startup times loading the whole library.

To illustrate this inefficiency pattern, we examine Chameleon. Chameleon is a Python HTML template engine that we imported into a serverless HTML rendering application. Running the application showed that *pkg_resources*, one of the libraries it depends on, took a significant time to import. The Chameleon version that was used to deploy the application was 4.1.0, when checking the GitHub repository of the application, we observed that the developers pushed an update where they dropped the *pkg_resources* dependency because of its effect of high startup time. The update did not change the behavior of the application.

This scenario highlights how library choice uncovers a new inefficiency pattern. When the

developers choose libraries to run a serverless application on top of, they are presented with a list of library choices to develop with. This inefficiency pattern discusses how that choice can impact startup time. When faced with this inefficiency, developers can choose to replace the implementation of features with more lightweight alternatives.

CHAPTER 4

Design and Implementation

This section introduces $ColdSpy$, a diagnostic tool designed to profile Python applications deployed in serverless environments. $ColdSpy$ is a dynamic program analysis tool written as an importable Python module. The tool's main objective is detecting libraries that cause significant initialization time and then identifying the frequency and code location of their utilization. We verify that developers can use $ColdSpy$ to guide code inefficiency optimizations that bring responses in serverless applications down.

$ColdSpy$'s approach boils down to profiling the Python application in two dimensions:

- The time duration a module takes to import as a metric of its impact on the cold-start time of the serverless application.
- Number of samples coming from the module as an indicator metric of the utilization and relevance of the module to the application runtime.

In addition to the number of samples, the stack traces of the samples are also collected to generate a context calling tree (CCT) of the Python application runtime allowing the inspection of where in the code the sample is coming from and being used.

$ColdSpy$ is implemented as a Python module imported at the beginning of the target Python application before any other imports. Using the built-in signal library of Python, the runtime of the application is profiled with a set time interval of time (e.g. 5ms).

```
1 import signal
2 ...
```

```
interval = 0.005
signal.signal(signal.SIGPROF, _sample)
signal.setitimer(signal.ITIMER_PROF, interval, interval)
```

Listing 4.1: Every 5ms a sample function will be invoked having the current stack frame passed to it.

```
import traceback
...
tree = cct.CCT()

def _sample(_, frame):
    stack = traceback.extract_stack(frame)
    # ignore if sample is coming from the profiler
    for f in stack:
        if f.filename == PROFILER_FILEPATH:
            return
    tree.add_sample(stack)
```

Listing 4.2: Retrieving the stack from the frame and adding to the CCT

The function retrieves the stack that the frame comes from which is a list of the frames leading up to the sampled one. It will ignore the stack if the frame originates from the profiling mechanism and lastly, it will add it to the sampling data structure that is the CCT. Listing 4.2 shows how the samples are added to the tree.

To collect information on what imports have been made in the application, the profiling module makes use of a Python option that logs the name of every import module along with the time it takes to import. This option is built into the main implementation of CPython, used in AWS Lambda. It is enabled through by setting the environment variable *PYTHONPROFILEIMPORTTIME* to any value. Enabling this option will cause the running Python process to start outputting to standard error the name of a module and the time it took by itself titled self-time and the time it took to import nested imports recursively as shown in Listing 4.3.

```
user@machine ~ % python
import time: self [us] | cumulative | imported package
import time:       216 |        216 |   _io
import time:        45 |         45 |   marshal
import time:       330 |        330 |   posix
import time:       773 |       1363 | _frozen_importlib_external
...
```

Listing 4.3: Running the python interpreter with the *PYTHONPROFILEIMPORTTIME* environment variable

Redirecting standard error in a Python application can be a straightforward thing to do. If the output originates from the Python code itself, developers can easily capture and redirect output using *sys.stderr*. However the challenge arises as overriding the *sys.stderr* function will not redirect output coming from C Code, as it is being executed within the Python process.

The C runtime uses its own mechanisms to handle output, writing *stderr* through file descriptors at a lower level than Python's *stderr* stream, bypassing the *sys.stderr* function.

To overcome this challenge, the underlying file descriptors are manipulated using the built-in *os* library as shown in Listing 4.4. By creating a temporary file and redirecting the stderr file descriptor to it, all output generated during the process execution will be captured [6].

```
original_stderr_fd = sys.stderr.fileno()
tfile = tempfile.TemporaryFile(mode='w+b')
libc = ctypes.CDLL(None)
libc.fflush(ctypes.c_void_p.in_dll(libc, 'stderr'))
sys.stderr.close()
os.dup2(to_fd, original_stderr_fd)
sys.stderr = io.TextIOWrapper(os.fdopen(original_stderr_fd, 'wb'))
```

Listing 4.4: Capturing C stderr output by redirecting it to a created temporary file descriptor

To this point, it was explained how the desired profiling data is collected. To analyze the data, a developer firstly needs access to it. The data is exported to a location accessible by the application

Application: streamalert

Global Stats: {"samples":86689,"init":2170100,"exec":0,"execCount":0,"fileCount":21}

Choose view

CCT

Module Tree by Dot

Dynamic Import Tree

Hide import samples ☐

Only show import samples ☐

Module filter: null

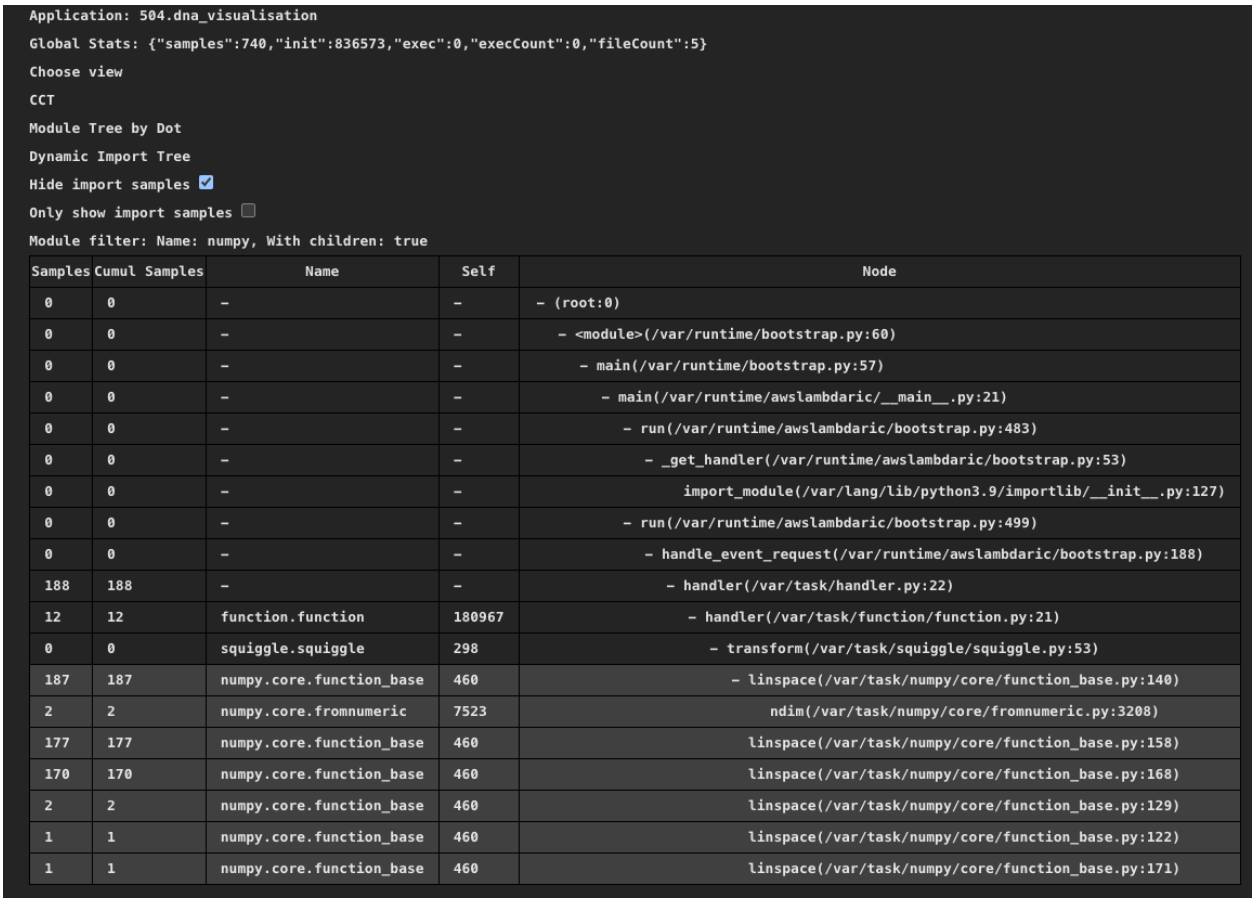
	Node	Score	Samples	Cumulative Samples by Dot	Self	Cumulative Self by Dot	Cumulative Time by Import	File
+	policyuniverse	99.9	3	57	997815	1109012	1053372	/var/task/policyuniverse/__init__.py
+	botocore	19.1	1	223	2705	212306	2705	/var/task/botocore/__init__.py
	pathlib2	18.5	1	1	204844	204844	207548	/var/task/pathlib2/__init__.py
+	publishers	10.9	0	0	22338	120480	22338	/var/task/publishers/__init__.py
+	boto3	9.7	0	36	1257	108016	432342	/var/task/boto3/__init__.py
+	urllib3	7.0	0	433	1634	78843	79714	/var/task/urllib3/__init__.py
+	netaddr	4.6	0	13	1609	50565	51615	/var/task/netaddr/__init__.py
+	streamalert	4.1	0	47039	0	90969	0	WARNING: module does not exist

Figure 4.1: Each row in the table is a library name with its import times and samples coming from it

developer. In an AWS Lambda environment, the data will be bundled in a pickle (*.pkl*) data format file and exported to S3 which happens every time the function returns a response.

After acquiring the data, multiple views of the data can be established. One example view is the grouping of the modules by prefix before the first dot as shown in Figure 4.1. It is possible to show the duration a library takes by summing up the import times of its sub-modules. The calling context tree is another constructed view as seen in Figure 4.2 where all the sampled stack frames are merged into one tree. After locating a library of interest to optimize, the CCT could be inspected to know which call paths are being traversed to reach the samples.

In this chapter, we devise $ColdSpy$ and show that it is possible to collect profiling data that ranks imported modules with a score on how likely they can be optimized and offers a CCT view that increases the degree of helping the developer with debugging and finding inefficiencies in their code.



Application: 504.dna_visualisation

Global Stats: {"samples":740,"init":836573,"exec":0,"execCount":0,"fileCount":5}

Choose view

CCT

Module Tree by Dot

Dynamic Import Tree

Hide import samples ☑

Only show import samples ☐

Module filter: Name: numpy, With children: true

Samples	Cumul Samples	Name	Self	Node
0	0	-	-	- (root:0)
0	0	-	-	- <module>(/var/runtime/bootstrap.py:60)
0	0	-	-	- main(/var/runtime/bootstrap.py:57)
0	0	-	-	- main(/var/runtime/awslambdaric/__main__.py:21)
0	0	-	-	- run(/var/runtime/awslambdaric/bootstrap.py:483)
0	0	-	-	- _get_handler(/var/runtime/awslambdaric/bootstrap.py:53)
0	0	-	-	import_module(/var/lang/lib/python3.9/importlib/__init__.py:127)
0	0	-	-	- run(/var/runtime/awslambdaric/bootstrap.py:499)
0	0	-	-	- handle_event_request(/var/runtime/awslambdaric/bootstrap.py:188)
188	188	-	-	- handler(/var/task/handler.py:22)
12	12	function.function	180967	- handler(/var/task/function/function.py:21)
0	0	squiggle.squiggle	298	- transform(/var/task/squiggle/squiggle.py:53)
187	187	numpy.core.function_base	460	- linspace(/var/task/numpy/core/function_base.py:140)
2	2	numpy.core.fromnumeric	7523	ndim(/var/task/numpy/core/fromnumeric.py:3208)
177	177	numpy.core.function_base	460	linspace(/var/task/numpy/core/function_base.py:158)
170	170	numpy.core.function_base	460	linspace(/var/task/numpy/core/function_base.py:168)
2	2	numpy.core.function_base	460	linspace(/var/task/numpy/core/function_base.py:129)
1	1	numpy.core.function_base	460	linspace(/var/task/numpy/core/function_base.py:122)
1	1	numpy.core.function_base	460	linspace(/var/task/numpy/core/function_base.py:171)

Figure 4.2: An example CCT filtered by Numpy in a DNA Visualization application

CHAPTER 5

Evaluation and Case Studies

We study the effectiveness of $ColdSpy$ on three real-world AWS Lambda applications evidenced to be highly regarded within the developer community by their substantial number of stars on GitHub. These applications serve as great candidates to demonstrate $ColdSpy$'s capabilities in identifying and mitigating performance bottlenecks specific to serverless deployments.

Table 5.1 displays how each application had a number of categorization inefficiencies identified in it and optimized with the improvement in cold-start time as a percentage.

Application	Remove Library	Lazy Load	Remove Feature	Replace Library	Cold-start reduction
StreamAlert	✓	✓			42%
OCRmyPDF		✓	✓		38%
Chameleon				✓	14%

Table 5.1: The optimizations applied to the applications depending on the application names

The evaluation is structured by conducting case studies on applications with real-world use cases of a serverless environment.

5.1 StreamAlert

This case study examines StreamAlert, an example of how library loading incurs high cold-start time. StreamAlert is a serverless, real-time data analysis framework that can be configured to receive large amounts of data from any source, analyze it, and send alerts according to user-defined rules. Organizations can use this framework to detect anomalies or suspicious activity in data streams, promptly sending an alert whenever any of the defined rules activate on received object data. The

project uses Python, making it easier for developers to add rules, edit them, and use any Python libraries in them [1].

StreamAlert comes with extensive documentation containing instructions for deployment. The project code uses Infrastructure-as-Code (IaC) at its core allowing for ease of deployment. IaC is used because in addition to the serverless Lambda functions, other deployed services range from S3 to DynamoDB and Athena on AWS. Among the deployed Lambda functions is the *rules-engine* function that does the rule evaluation on the logs. When configuring the project, the developer can specify which libraries the code would depend on, two of which, for example, are libraries *PolicyUniverse* and *pathlib2*.

After looking for a real-world to run the application on, an online dataset of anonymized AWS CloudTrail logs is chosen as the workload data [22]. The dataset is separated into parts and uploaded to S3 triggering the StreamAlert workflow. Throughout the workflow, data will eventually reach the *rules-engine* function. *rules-engine* will evaluate multiple rules on the logs and check if any of them trigger an alert. One key characteristic of the Python script rule files executed by *rules-engine* is their dependence on other libraries. The rule function calls these libraries accordingly after a specific sequence of operations as directed by the input data to the rule. This indicates that a rule file will only call a library if input log data is of a specific type that might or might not be a frequent occurrence in the dataset during the life of the application.

The initialization of the execution environment includes the import of two libraries, *PolicyUniverse* and *pathlib2*. *PolicyUniverse* is a library available from PyPI (Python Package Index) used by one of the rules to parse AWS *IAM* and Resource Policies [16, 20].

After running the dataset by the *rules-engine* instance, cold-start time is observed to take a significant part of the function runtime. This performance detail of the function indicates that profiling the function could give insight into where this time is coming from.

Profiling StreamAlert with $ColdSpy$, shows that a library called *PolicyUniverse* is at the top of that list taking $\sim$1.2 seconds to be imported. The CCT also reports the number of samples coming from *PolicyUniverse* in particular.

After collecting around 90 thousand samples in the Python code of the Rules Engine lambda function, only 57 of them came from the *PolicyUniverse* library. This profile indicates that *PolicyUniverse* is a module that has a high import time-to-sampling ratio, flagging it as a high-priority candidate for optimization.

By following the call paths in the CCT we can trace the usage of PolicyUniverse to a rule file in the code-base. In the code of that rule, *PolicyUniverse* is only called when the log has a specific field value.

```
"""Alert when resources are made public."""
import json
from policyuniverse.policy import Policy
from streamalert.shared.rule import rule

@rule(logs=['cloudtrail:events'])
def cloudtrail_public_resources(rec):
    ...
    if rec['eventName'] == 'PutBucketPolicy':
        policy = rec.get('requestParameters', {}).get('bucketPolicy', None)
        if not policy:
            return False
        policy = Policy(policy)
        if policy.is_internet_accessible():
            return True
    ...
```

Listing 5.1: The rule file in the Rules Engine code that imports PolicyUniverse

However, in the log data, this log field value is not very common, which suggests that importing this library at every invocation of the Lambda may not be needed as for some invocations the library is not called.

As an experiment, we isolated the code of the rule and executed it on all of the logs of the dataset. We perform an analysis on the rule and observe the number of times the library is called,

and the number of times it is imported but not called, wasting unneeded cold start time.

The analysis is structured by first referring to the duration differences of the incoming logs' timestamps. If the wait is more than 5 minutes then it will cause a cold start and a cold-start counter is increased. For each of those cold starts, we record whether the library was used or not. The number of cold starts in which the library was not used is the time that was wasted which could be optimized.

```
from datetime import datetime, timedelta
from cloudtrail_public_resources import cloudtrail_public_resources

coldstarts_num, warmstart_num, loaded_cold, loaded_warm = 0, 0, 0, 0, 0

current = datetime.fromisoformat(records[0]['eventTime'][:-1])
for r in records[1:]:
    nextr = datetime.fromisoformat(r['eventTime'][:-1])
    loaded = cloudtrail_public_resources(r)
    coldstart = nextr - current > timedelta(minutes=ttl)
    if coldstart:
        coldstarts_num += 1 # increase the cold start time
        if loaded: loaded_cold += 1 # loaded on cold start
    else:
        warmstart_num += 1 # increase the warm start amount
        if loaded: loaded_warm += 1  # loaded on a warm start
    current = nextr
```

Listing 5.2: Processing the CloudTrail logs to count the number of cold-starts generated

This analysis estimates that the number of freshly initialized instances is ∼50K. Only in 79 of those, however, does the application use the loaded library *PolicyUniverse*. This demonstrates an extreme under-utilization of the library and the opportunity to load it on demand, saving ∼1.2 seconds in each of the remaining cold starts outside the 79 ones that do not need the library.

Multiplying 1.2 seconds by the number of function instances that did not load the library after the optimizations results in 60K seconds that were saved by the simple action of the developer

moving the import statement from the top of the file to a line just before it is needed.

```
@rule(logs=['cloudtrail:events'])
def cloudtrail_public_resources(rec):
    ...
    if rec['eventName'] == 'PutBucketPolicy':
        policy = rec.get('requestParameters', {}).get('bucketPolicy', None)
        if not policy:
            return False
        from policyuniverse.policy import Policy # line change
        policy = Policy(policy)
        if policy.is_internet_accessible():
            return True
    ...
    return False
```

Listing 5.3: Lazy loading by moving the library loading to a line right before it is being called

As a solution, Listing 5.3 shows how we implemented a technique known as lazy loading by moving the import statement from the top of the file to just right before it is needed. The code change results in the library only being loaded when it is necessary for the execution, rather than during the runtime of every function instance. Only a small number of function instances actually load the library during their execution lifetime, which greatly reduces the cold-start for the majority of the function instances.

Figure 5.1 shows the rate of improvement of the Lambda function after performing the code changes. On average, this optimization resulted in 42% and 39% time reduction in initialization and end-to-end respectively.

In conclusion, StreamAlert serves as a good example of how profiling serverless Python applications could uncover the existence and the nature of inefficiencies of under-utilized imports, guiding the optimization needed to reduce cold starts.

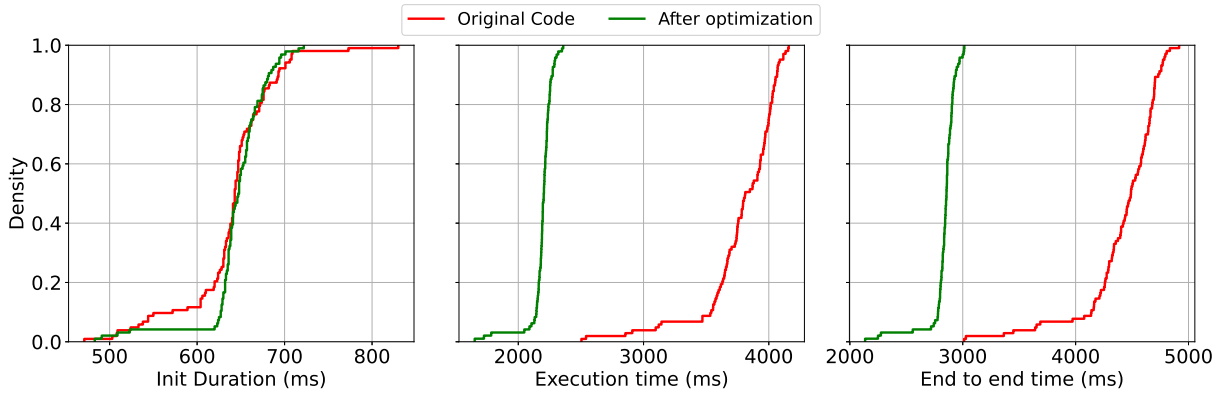


Figure 5.1: The CDF of the initialization, execution, and end-to-end duration after performing 100 invocations of *rules-engine* from a cold state.

5.2 OCRmyPDF

This case study examines a serverless Python application that uses the *OCRmyPDF* library. *OCRmyPDF* is an open-source and highly popular Python library that allows running OCR (Optical Character Recognition) technology on PDFs. The tool then adds a top layer of the PDF containing select-able text allowing the PDFs to be searched by other processing workflows.

The ability to ingest and process variable streams of PDF documents is a highly suitable use case for a serverless environment. Institutions of all kinds deal with huge numbers of documents. Considering a legal firm that deals with case files, contracts, and legal precedent; Being able to search previous documents and digital archives could greatly facilitate case preparation and research.

Deploying *OCRmyPDF* to AWS Lambda can prove to be a challenge as it depends on *Tesseract* and *Ghostscript* binaries. Although these binaries come pre-built for the known Linux distributions, this is not the case for Amazon Linux, the operating system that runs on Lambda functions. However, building these binaries on any Amazon Linux instance beforehand and packaging them in the Lambda function package does the trick [3, 15, 24].

Profiling the Lambda application with $ColdSpy$ shows that two libraries *PDFMiner* and *Rich* that *OCRmyPDF* depends on have high import times while not having lots of samples coming from their package modules.

The *PDFMiner* library shows $ColdSpy$ to have very few samples coming from it suggesting that it is not being called throughout the execution of the application. Upon inspection of the code to see where the library is being called, it was noted that from the *OCRmyPDF* package, there is only one file importing *PDFMiner* modules and only using those to define a function *get_page_analysis*. The *get_page_analysis* function is in turn only being called under an if condition that checks for a *detailed_analysis* variable initialized to *False*.

```
class PageInfo:
    """Information about type of contents on each page in a PDF."""
    def _gather_pageinfo(...):
        ...
         if check_this_page and detailed_analysis:
            pscript5_mode = str(pdf.docinfo.get(Name.Creator)).startswith('PScript5')
            miner = get_page_analysis(infile, pageno, pscript5_mode)
            self._textboxes = list(simplify_textboxes(miner, get_text_boxes))
            ...
```

Listing 5.4: The *get_page_analysis* call that is only performed under an if condition that is not visited by default

```
def get_page_analysis(infile, pageno, pscript5_mode):
    """Get the page analysis for a given page."""
    rman = PDFMiner.pdfinterp.PDFResourceManager(caching=True)
    disable_boxes_flow = None
    dev = TextPositionTracker(... )
    interp = PDFMiner.pdfinterp.PDFPageInterpreter(rman, dev)
    ...
    return dev.get_result()
...
```

Listing 5.5: The *get_page_analysis* function definition showing calling the *PDFMiner* library

This analysis shows that the *PDFMiner* library could be simply lazy-loaded by moving the import of the `get_detailed_analysis` function to the line right before the one it is needed

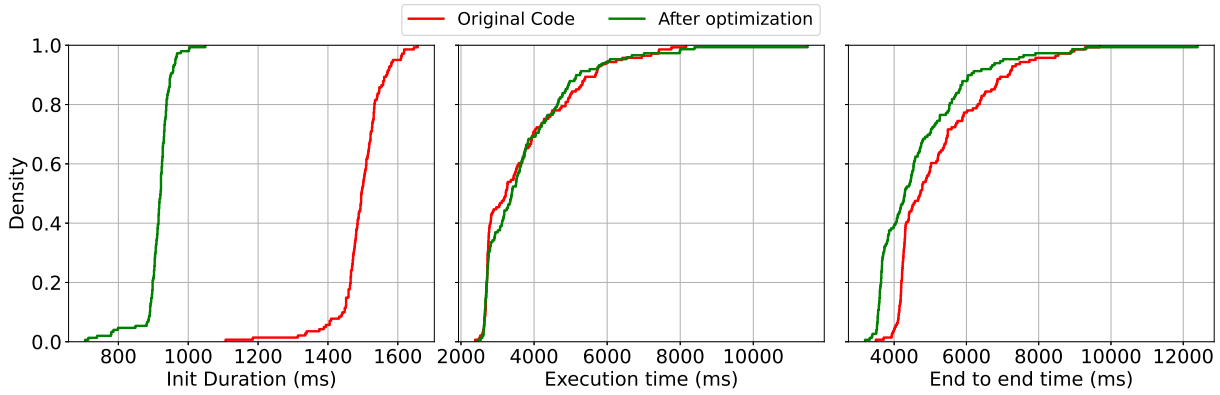


Figure 5.2: The CDF of the initialization, execution, and end-to-end duration after performing 100 invocations of the *OCRmyPDF* Lambda application from a cold state.

in.

Another library that the tool picked up is *Rich*. "Rich is a Python library for rich text and beautiful formatting in the terminal." [18]. Which suggests that the application is logging into the console. When checking the console it could be observed that there is indeed a progress bar being logged with the help of the *Rich* library.

Progress bars when dealing with a single PDF file with a big size from a user perspective running the program from their terminal are an appreciated user experience. However, in a serverless AWS Lambda setting that is running batch processes, a progress bar would have little benefit and therefore can be avoided, a compromise that brings a much sought-after benefit which is a cold-start time reduction.

This optimization is categorized as aggressive. Although the computational goal of the Lambda function of processing the PDF file is achieved, its behavior is slightly changed.

The improvement in performance for the *OCRmyPDF* application after removing the *Rich* library is captured in Figure 5.2. The optimization results in a reduction in initialization and end-to-end time by 38% and 10% respectively.

This case study serves as a great example of how initialization times in Python come from library imports that could be avoided. As well as of how the guidance of the $ColdSpy$ simplifies and accelerates the process for developers to find opportunities for optimizations in their serverless

applications.

5.3 Chameleon

In this part, we explore yet another application that demonstrates the utility of the developed profiling tool. Serverless functions can make a perfectly fine use case for serving a web application page, including the usage of an HTML template. One HTML templating tool is Chameleon, a "Fast HTML/XML template engine for Python" [9].

Chameleon is a relatively small library, without dependencies on any other libraries. When profiling this application with $ColdSpy$, however, *pkg_resources*, a library of interest showed up in the profiling analysis as a top candidate for optimization, taking high import time without considerable samples coming from it.

pkg_resources is part of the *setuptools* package [4], aiding in package discovery, version management, and resource access. *pkg_resources* provides a standard API for Python applications to discover and access package information and resources, such as files and data bundled within packages. This capability is essential for applications that rely on external libraries and assets, abstracting the complexity of the file system layout and package installation types.

The version of Chameleon that was used in this case study is 4.1.0 released on August 31, 2023. Subsequent versions have moved away from *pkg_resources* in favor of *importlib-resources*, a library that offers similar functionalities having more efficient implementation. [25]

The change from *pkg_resources* to *importlib-resources* within Chameleon illustrates a significant pattern in our study: optimization of a cold-start inefficiency through the replacement of one implementation with another.

Figure 5.3 shows the rate of improvement of the Lambda function after replacing *pkg_resources* with *importlib-resources*. On average, this optimization resulted in 14% and 5% time reduction in initialization and end-to-end respectively.

This highlights one more use-case of $ColdSpy$ of catching possibilities of replacing an imported Python module with another. As multiple implementations could achieved the same behavior by an application but incur different cold-start time, it is possible for a Python application to replace an

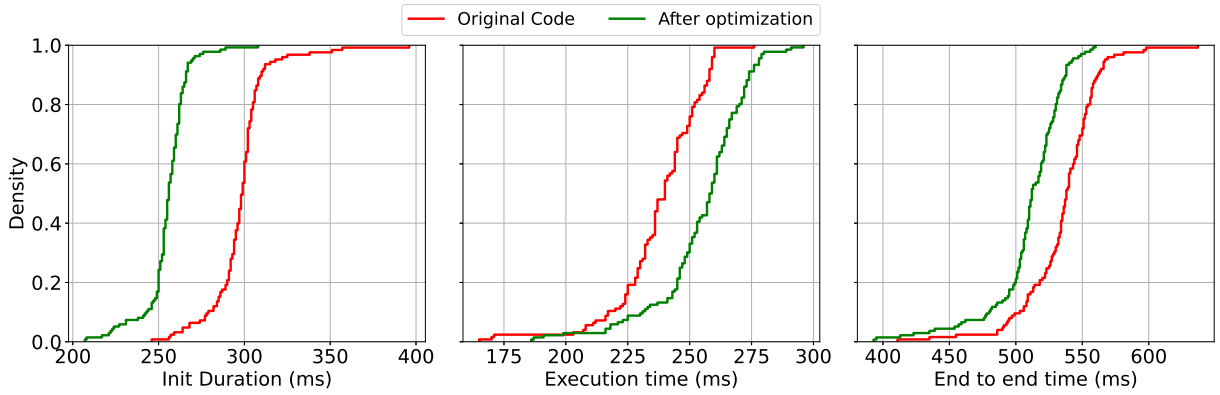


Figure 5.3: The CDF of the initialization, execution, and end-to-end duration after performing 100 invocations of the *Chameleon* Lambda application from a cold state.

inefficient import with another taking less cold-start time.

CHAPTER 6

Conclusion

While the serverless architecture enables on-demand scalability and cost savings, it introduces latency that significantly impacts the response time of its function instances. This research introduces a comprehensive approach to addressing the cold-start inefficiency in serverless applications running Python.

After doing case studies on popular, real-life examples of serverless applications running known Python packages, the research demonstrates the utility of a specialized profiling tool capable of identifying optimization opportunities within Python applications deployed as AWS Lambdas. $ColdSpy$ dynamically addresses the inefficiencies in module imports and call paths paths providing insights beyond what static analysis could uncover.

In this work, cold-start inefficiencies were categorized in four patterns: 1) Never used 2) Rarely used 3) Feature redundancy, and 4) Library choice. We propose specific optimization strategies for each category, helping developers prioritize which modules to optimize based on import time and the frequency of their use during the application execution runtime.

We validated the effectiveness of $ColdSpy$ by using it in real-world applications, leading to substantial reductions in cold-start time. The improvements enhance performance and result in cost savings for both serverless users and providers.

This research contributes to the ongoing improvement of serverless computing performance. By using $ColdSpy$ for dynamic analysis of their serverless applications, developers can better analyze and iterate on their serverless applications. The potential impact of this work is achieving more efficient serverless applications encouraging their further adoption and use in the industry.

BIBLIOGRAPHY

[1] AirBnB. Streamalert is a serverless, realtime data analysis framework... https://github.com/airbnb/streamalert.

[2] 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, 2018.

[3] Inc. Artifex Software. Ghostscript is a suite of software based on an interpreter for adobe systems' postscript and portable document format (pdf) page description languages. https://www.ghostscript.com/.

[4] Python Packaging Authority. Easily download, build, install, upgrade, and uninstall python packages. https://pypi.org/project/setuptools/.

[5] James R. Barlow. Ocrmypdf adds an ocr text layer to scanned pdf files, allowing them to be searched. https://github.com/ocrmypdf/OCRmyPDF.

[6] Eli Bendersky. Redirecting all kinds of stdout in python. https://eli.thegreenplace.net/2015/redirecting-all-kinds-of-stdout-in-python/.

[7] Emery D Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with {SCALENE}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 51–64, 2023.

[8] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 153–167, 2021.

[9] Malthe Borch. Fast html/xml template engine for python. https://github.com/malthe/chameleon.

[10] 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*, pages 467–481, 2020.

[11] Hanfei Yu et. al. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2024.

[12] Python Software Foundation. The python programming language. `www.python.org/`.

[13] Ben Frederickson. Sampling profiler for python programs. `https://github.com/benfred/py-spy`.

[14] 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*, pages 386–400, 2021.

[15] Benjamin Genz. A layer for aws lambda containing the tesseract c libraries and tesseract executable. `https://github.com/bweigel/aws-lambda-tesseract-layer`.

[16] Jazzband. Backport of pathlib aiming to support the full stdlib python api. `https://pypi.org/project/pathlib2/`.

[17] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–29, 2023.

[18] Will McGugan. Rich is a python library for rich text and beautiful formatting in the terminal. `https://pypi.org/project/rich/`.

[19] Ron Miller. AWS Lambda Makes Serverless Applications A Reality — TechCrunch — techcrunch.com. `https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/`, 2015. [Accessed 01-04-2024].

[20] Netflix-Skunkworks. Parse and process aws iam policies, statements, arns, and wildcards. `https://github.com/Netflix-Skunkworks/policyuniverse`.

[21] 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, 2018.

[22] Scott Piper. Public dataset of cloudtrail logs from flaws.cloud. `https://summitroute.com/blog/2020/10/09/public_dataset_of_cloudtrail_logs_from_flaws_cloud/`, 2020. [Accessed: 2024-03-28].

[23] 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*, pages 714–729, 2022.

[24] The Tesseract OCR Team. Tesseract ocr is an open source optical character recognition engine for various operating systems. https://github.com/tesseract-ocr/tesseract.

[25] Barry Warsaw. Read resources from python packages. https://pypi.org/project/importlib-resources/.