FaaSLight: General Application-level Cold-start Latency Optimization for Function-as-a-Service in Serverless Computing

XUANZHE LIU and JINFENG WEN, Peking University, China
ZHENPENG CHEN, University College London, United Kingdom
DING LI and JUNKAI CHEN, Peking University, China
YI LIU, Advanced Institute of Big Data, Beijing, China
HAOYU WANG, Huazhong University of Science and Technology, China
XIN JIN, Peking University, China

Serverless computing is a popular cloud computing paradigm that frees developers from server management. Function-as-a-Service (FaaS) is the most popular implementation of serverless computing, representing applications as event-driven and stateless functions. However, existing studies report that functions of FaaS applications severely suffer from cold-start latency.

In this article, we propose an approach, namely, *FaaSLight*, to accelerating the cold start for FaaS applications through application-level optimization. We first conduct a measurement study to investigate the possible root cause of the cold-start problem of FaaS. The result shows that application code loading latency is a significant overhead. Therefore, loading only indispensable code from FaaS applications can be an adequate solution. Based on this insight, we identify code related to application functionalities by constructing the function-level call graph and separate other code (i.e., optional code) from FaaS applications. The separated optional code can be loaded on demand to avoid the inaccurate identification of indispensable code causing application failure. In particular, a key principle guiding the design of *FaaSLight* is inherently general, i.e., *platform-* and *language-agnostic*. In practice, *FaaSLight* can be effectively applied to FaaS applications developed in different programming languages (Python and JavaScript), and can be seamlessly deployed on popular serverless platforms such as AWS Lambda and Google Cloud Functions, without having to modify the underlying OSes or hypervisors, nor introducing any additional manual engineering efforts to developers. The evaluation results on real-world FaaS applications show that *FaaSLight* can significantly reduce the code

119

loading latency (up to 78.95%, 28.78% on average), thereby reducing the cold-start latency. As a result, the total response latency of functions can be decreased by up to 42.05% (19.21% on average). Compared with the state-of-the-art, *FaaSLight* achieves a 21.25× improvement in reducing the average total response latency.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Additional Key Words and Phrases: Serverless computing, cold start, performance optimization, optional function elimination

1 INTRODUCTION

Serverless computing is a popular cloud computing paradigm and has been applied to various domains, including machine learning [67], scientific computing [97], and video processing [57]. It is predicted that 50% of global enterprises will employ serverless computing by 2025 [14]. To embrace this paradigm, major cloud vendors have rolled out various serverless platforms, such as AWS Lambda [26], Microsoft Azure Functions [30], and Google Cloud Functions [34]. In these serverless platforms, **Function-as-a-Service (FaaS)** is the most prominent implementation pattern [55, 77, 84, 102, 103, 108]. FaaS represents event-driven and stateless functions (*serverless functions*). Developers can implement their applications as a combination of serverless functions, and these applications are called FaaS applications.

The underlying serverless platforms automatically handle resource management. Therefore, developers do not need to manage servers or VM instances to run FaaS applications. Instead, serverless functions are dynamically allocated with resources when they are triggered by events (e.g., an HTTP request). If a serverless function has not been used for a while, then the platform will release the resources. In this way, resource management can be lightweight and efficient.

However, such on-demand resource management in serverless computing introduces the cost of longer response latency. The resources of idle serverless functions will be released. Therefore, the serverless platform has to initialize the execution environment for the invocations to functions that are not frequently used, which prolongs the functions' response latency. In practice, the latency spent on preparing the execution environment (cold-start latency for short) has been demonstrated to be significant in the total response latency in various application scenarios, such as IoT applications [88, 113], video processing [57, 69], and machine learning [61, 67]. For example, a previous study [70] reported that the cold-start latency could be as much as 80% of the total response latency. Therefore, optimizing cold-start latency is a critical challenge of contemporary FaaS applications.

Several efforts have been made to optimize the cold-start latency at the *system level* (i.e., optimizing the underlying serverless platforms), such as developing lightweight virtualization technology of containers [64], adjusting the scheduling policy to keep instances warm [106], and redesigning sandbox runtime mechanisms [55, 85]. Although these efforts are demonstrated to be efficient and promising, they all inherently require extensive engineering efforts to modify underlying OSes or hypervisors. Serverless platform vendors should have concerns about adopting and implementing substantial changes to their existing infrastructures. In addition, they also have concerns about

security mechanisms, e.g., ASLR [64]. To the best of our knowledge, none of the aforementioned techniques have been applied to commercial serverless platforms.

In this article, we focus on applications built on FaaS. Compared to system-level optimization of cold-start latency, which requires intrusive changes to the underlying platforms, we aim to tackle this problem at the application level. Our guiding principle is to provide a *platform/language-independent* and *developer-free* technique that application developers can adopt to optimize the cold-start latency of serverless functions on most existing platforms. To achieve our goal, we first investigate the possible root cause of the cold-start overhead of FaaS applications. We conduct a measurement study on real-world FaaS applications. The result shows that the application code loading latency takes a significant part of the cold-start latency.

Based on this observation, we propose an application-level approach, namely, *FaaSLight*, to optimize the cold-start latency of serverless functions by reducing the size of executed code, i.e., loading only necessary code. One key insight of *FaaSLight* is to identify code related to application functionalities (called *indispensable code*) through constructing the function-level call graph. Then, it separates other code (called *optional code*) from the original application by analyzing the intermediate representation of the application code. To guarantee the correctness of the application, *FaaSLight* does not remove the optional code, but compresses the optional code into a lightweight file and fetches the separated code in an on-demand loading way if the code is invoked. In this way, we can reduce the code size in the loading process and guarantee the correctness and availability of the final FaaS applications.

It is worth mentioning two key design principles raised by *FaaSLight*. First, *FaaSLight* presents the support of *generalizability*. That is to say, the principled design behind *FaaSLight* itself is independent of underlying heterogeneous serverless platforms and programming languages of the FaaS. As demonstrated later in this article, *FaaSLight* can be easily applied to FaaS applications that are developed in different programming languages (i.e., Python and JavaScript) and executed on different platforms (i.e., popular AWS Lambda and Google Cloud Functions). Second, *FaaSLight* is *developer-free* to existing (or legacy) FaaS applications. Inherently, *FaaSLight* does not introduce **any additional** manual efforts for developers, nor requires any modification of existing serverless platforms. In practice, *FaaSLight* can be simply deployed as a service on current serverless platforms. When developers upload their applications, *FaaSLight* can process the code optimization and application deployment automatically. In other words, developers are unaware of any changes when *FaaSLight* works.

We implement *FaaSLight* as the Python and JavaScript prototypes, since they are the most widely used languages in the serverless community. We evaluate its effectiveness on real-world FaaS applications. The results show that *FaaSLight* can reduce the application code loading latency by up to 78.95% (on average 28.78%), thereby reducing the cold-start latency. As a result, the total response latency of serverless functions can be decreased by up to 42.05% (on average 19.21%). As an additional benefit, *FaaSLight* can decrease the runtime memory of serverless functions by up to 58.82% (on average 14.79%) due to the reduced size of loaded code. Compared with the state-of-the-art, *FaaSLight* achieves a 21.25× improvement in reducing the average total response latency.

To the best of our knowledge, *FaaSLight* is the first application-level effort to optimize the cold-start latency of serverless functions. To summarize, this article makes the following contributions.

- We conduct a measurement study to demystify the possible root cause of the cold-start latency of serverless functions. The result shows that the application code loading latency is the significant overhead.
- We propose a general application-level performance optimization approach to reducing the cold-start latency without compromising the effectiveness, correctness, and availability of FaaS applications.

- We evaluate our approach using real-world FaaS applications under heterogeneous settings (different programming languages and platforms), and the results show that it can significantly reduce cold-start latency. We demonstrate that our approach, i.e., loading only indispensable code, can significantly improve the performance of FaaS applications.

We have implemented the prototype of *FaaSLight* to support both Python and JavaScript. The code and scripts have been open-sourced on GitHub.[1]

2 BACKGROUND

In this section, we introduce the background of serverless computing and then describe the cold-start problem.

2.1 Serverless Computing

Serverless computing allows software developers to efficiently develop and deploy applications to the market without having to manage the underlying infrastructure [107, 109, 110], i.e., "serverless" means no server management for developers. In serverless computing, FaaS is the prominent and widely adopted implementation [55, 77, 102, 107, 109]. Developers can rely on the FaaS fashion to focus solely on the business logic of applications, which are composed of a set of serverless functions. Such applications are called FaaS applications.

In FaaS applications, serverless functions and their dependency libraries are packaged into a single bundle, and then deployed to serverless platforms. If the application size exceeds the deployment restriction (e.g., 250 MB uncompressed size on AWS Lambda), then developers can deploy applications using container images with larger sizes [13]. After successful deployment, serverless functions will be triggered with predefined events, e.g., an HTTP request, a file update of cloud storage, or a timer going off. Once serverless functions are triggered, the serverless platform automatically allocates and launches dedicated function instances (e.g., VMs or containers) with restricted resources (e.g., CPU and memory) for them to execute their functionalities. When there are no incoming requests, launched instances and resources are later automatically released.

The invocation to a serverless function may go through two modes, the *cold-start mode* and the *warm start mode*. If the invoked function has not been used for a threshold (keep-alive time), then the invocation is in the cold-start mode. In this mode, the serverless platform needs to prepare new VMs or containers, transmit the code of the function from remote cloud storage (such as AWS S3 [104, 111]) to instances over the network, load the required code to initiate the application process, and finally execute the serverless function. On the contrary, if the invoked function is recently used (e.g., within 7 minutes for AWS Lambda [17]), the invocation is in the warm start mode, where the serverless platform reuses the launched instances of the same function.

This article focuses on the cold-start latency problem. For better illustration, we compare the cold-start latency with the warm-start latency in Figure 1.[2] The cold-start latency consists of three parts: the latency of preparing VMs or containers for the serverless function (instance initialization), transmitting the application over the network (application transmission), and loading the code related to the application (application code loading). We call instance initialization and application transmission as the preparation phase, and application code loading as the loading phase.

In contrast, the warm-start latency includes only the latency of scheduling reused instances, which is called the scheduling phase in our study.

---

[1]https://github.com/WenJinfeng/FaaSLight.

[2]Figure 1 shows key latencies, not showing fine-grained latencies like request reception and return due to the black-box feature of commodity platforms.
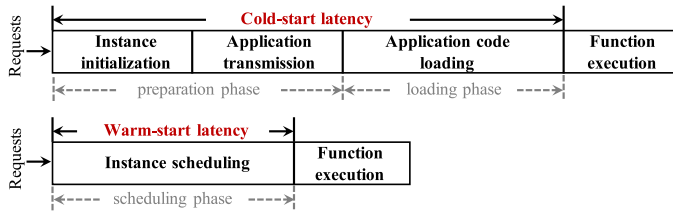
Fig. 1. The latency breakdown after requests come in cold and warm starts.

2.2 The Cold-start Problem

The cold-start latency significantly affects the overall runtime efficiency of a serverless function, because (1) the cold start happens frequently [55, 70, 85, 96, 105], and (2) once it happens, its latency takes a significant part of the end-to-end response latency of a serverless function [64, 70, 96, 101, 107, 111]. Because serverless functions often execute in only a few milliseconds, cold-start latency is significant in comparison. Fuerst et al. [70] showed that the cold-start latency could be as much as 80% of the total response latency. Generally, serverless functions are short-lived. Du et al. [64] calculated the ratio of function execution latency to total response latency for 14 serverless functions, and found that 12 serverless functions even cannot achieve 30%, emphasizing that the total response latency of a serverless function is dominated by its startup time. Singhvi et al. [101] also found that 57% of serverless functions have an execution time of less than 100 ms. Therefore, a fast cold start is critical for developers, because their tasks are often short-lived and completed quickly [64, 72, 78, 81].

3 MEASUREMENT STUDY

In this article, we focus on applications built on FaaS. A FaaS application is composed of serverless functions. To further investigate the possible root cause of the cold-start latency, we conduct a measurement study on real-world FaaS applications executed on AWS Lambda, which is the most popular and widely used serverless platform [20, 21].

3.1 Benchmarks

We select real-world FaaS applications from GitHub as benchmarks according to the following criteria. In the FaaS application, serverless functions and required dependency libraries are generally bundled together to deploy and execute. A FaaS application is selected when (1) it is written in Python, which is one of the most widely used languages in the serverless community [5], (2) its application code contains more than 20k lines of code, indicating a median and large application [94], (3) it has detailed instructions to guide us to execute it successfully, and (4) it is not a development tool, such as AWS SAM CLI [29], which is a command-line interface for developing serverless functions. Our final benchmarks consist of 15 real-world FaaS applications, ranging from data processing to machine learning. Specific details are shown in Table 1. In our study, for each FaaS application, its application size, the number of functions contained in the application, and the number of lines of code are denoted as *Size*, *FC*, and *LoC*, respectively. *FC* is calculated by analyzing the intermediate representation of code and recognizing the number of function definitions, while *LoC* is calculated by counting executable statements excluding single-line, multi-line, and document comments. As part of the serverless function, used dependency libraries are also involved in the calculations of *Size*, *FC*, and *LoC*. On average, our benchmarks have 349.70 MB *Size*, 45.42k *FC*, and 160.77k *LoC*.

Table 1. Details of our Benchmarks

App ID	Name	Size (MB)	FC (k)	LoC (k)	Description
App1	image-resize [1]	64.40	9.66	21.20	resize image and save to Boto3
App2	lambda-pillow [12]	85.91	11.56	30.68	import Boto3 and Pillow to test
App3	lambda-pandas [2]	83.88	41.76	151.40	use Pandas to converse data
App4	scikit-assign [10]	147.71	46.34	160.60	use Sklearn model to predict price
App5	lxml-requests [7]	25.26	6.57	47.18	use lxml to parse HTML pages
App6	pandas-numpy [9]	113.36	56.83	192.05	use Pandas and Numpy to generate data
App7	skimage-lambda [11]	261.47	72.63	267.94	download image and use Skimage to process
App8	opencv-pil [8]	152.96	35.20	93.71	use OpenCV and PIL to process image
App9	wine-ml-lambda [6]	248.25	81.91	317.88	train Sklearn model and predict wine quality
App10	lightgbm-sklearn [18]	221.70	56.45	216.92	use LightGBM model to predict
App11	sentiment-analysis [15]	240.92	73.45	280.46	use Sklearn model to predict sentiment statement
App12	tensorflow-lambda [4]	1,217.23	61.46	260.38	use TensorFlow regression model to predict data
App13	numpy-lambda [28]	70.22	37.69	99.12	use Numpy to converse matrix data
App14	lambda-opencv [3]	228.70	31.91	74.88	use OpenCV to get properties
App15	question-answering [16]	2,083.58	57.92	197.13	use Bert model to answer questions
Max		2,083.58	81.91	317.88	
Mean		349.70	45.42	160.77	

3.2 Measurement Results

We execute these FaaS applications in cold starts, and then obtain the preparation phase latency, loading phase latency, function execution latency (as described in Figure 1), as well as total response latency. The calculation of these latencies is as follows:

- *Loading phase latency* is the latency of the application code loading in cold starts. It can be extracted from the "Init Duration" attribute provided by AWS Lambda execution logs. It is also practical to use time breakpoints between import code statements.
- *Preparation phase latency* is the latency of the instance initialization and application transmission in cold starts. By setting time checkpoints at the request start and the beginning of the code body of serverless functions, we can obtain the overall cold-start latency including the preparation phase latency and loading phase latency. The preparation phase latency is extracted by removing the loading phase latency from the overall cold-start latency.
- *Function execution latency* is the latency of executing serverless functions. It can be extracted from the "Duration" attribute of AWS Lambda execution logs. Similarly, this latency can also be obtained by using time breakpoints at the beginning and end of serverless function code execution.
- *Total response latency* is the latency from request sending to request completion. By setting time checkpoints at the beginning and end of the request, the time interval is calculated as the total response latency.

We report the percentage of each latency to the total response latency for tested FaaS applications in Figure 2.[3] We observe that the loading phase latency is a significant overhead. Specifically, the cold-start latency, i.e., the sum of the preparation and loading phase latencies, takes up 88.70% of the total response latency, while the function execution latency is only 7.57%, on average. Particularly, for 12 FaaS applications, the function execution latency only takes less than 5% of the total response latency. We further analyze the latency percentage of two phases in the cold-start latency. On average, the loading phase latency is 46.24% of total response latency, while the preparation phase latency is 42.46%, as can be seen from Figure 2. It illustrates that the application code loading latency is a significant overhead in cold starts. Generally, the preparation phase latency contains

---

[3]App9 has two main functionalities, i.e., model training (App9-t) and model prediction (App9-p).
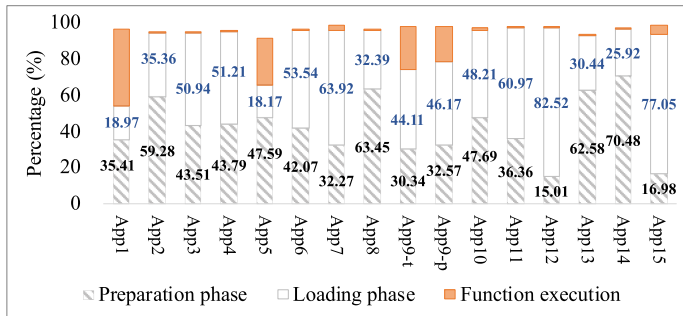
Fig. 2. The percentage of each latency to the total response latency.

the latency used to prepare VMs or containers for the serverless function (instance initialization) and transmit the application over the network (application transmission). Developers can hardly control the preparation phase latency, because commodity platforms such as AWS Lambda are not open to ordinary developers. The application code loading latency is caused by fetching application code in VMs or containers. In other words, reducing the code size of a FaaS application can optimize the application code loading latency.

FaaS applications are mostly written in high-level languages such as Python and JavaScript [5]. In practice, third-party dependency libraries are often imported to help serverless functions implement complex functionalities. Although developers usually use only a small subset of all supported functionalities of these libraries, all functionalities are still loaded completely [91, 92]. Thus, eliminating code not used by a FaaS application can potentially help optimize its application code loading latency. In practice, our previous experience demonstrated that a similar mechanism is effective to optimize client-side JavaScript applications [82, 83]. The application code loading latency is a significant part of the cold-start latency, which further affects the end-to-end response latency. Therefore, optimizing the code size of a FaaS application can potentially reduce its overall end-to-end latency. This insight motivates our proposed approach in this article.

4 APPROACH

In this section, we propose the design of *FaaSLight*, which optimizes the cold-start latency of FaaS applications by eliminating the optional functions. We define optional functions as those that cannot be reached from the entries of a FaaS application. Thus, we can remove optional functions without causing runtime errors from the FaaS application. *FaaSLight* accepts the source code of an existing FaaS application and outputs a modified version of the input application with optional functions removed. The key challenge for *FaaSLight* is to guarantee the correctness of optional function elimination while maximizing the reduction of cold-start latency. Serverless functions contained in the FaaS application are often built with dynamic languages, such as Python and JavaScript [5]. For such languages, it is particularly challenging, if not impossible, to identify the functions that can be reached from the entry of an invocation to the FaaS application with 100% accuracy. An overly aggressive solution that aims to identify more optional functions may misclassify non-optional functions as optional, leading to crashes in the runtime. On the flip side, an overly conservative solution may miss optional functions in the elimination process, leading to suboptimal cold-start time reduction. To address this challenge, *FaaSLight* adopts a combined strategy. It first leverages a static-analysis-based technique that identifies the functions that are reachable from the entries of a FaaS application. We define the identified functions as indispensable functions. The static analysis of *FaaSLight* is aggressive that aims to identify as many indispensable
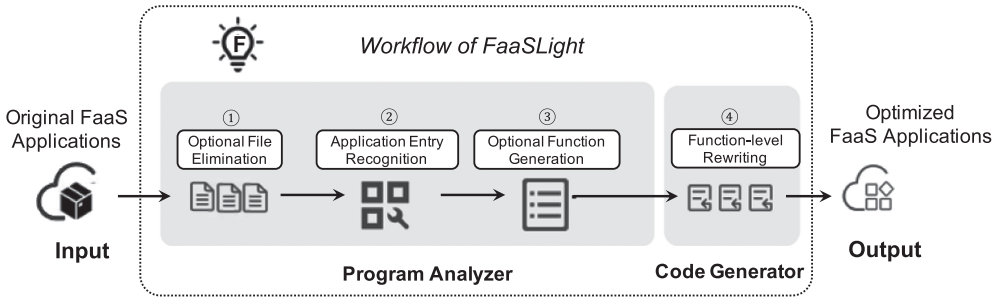
Fig. 3. The workflow overview of *FaaSLight*.

functions as possible. Then, after the static analysis, *FaaSLight* adopts a conservative on-demand loading strategy, which does not directly delete the optional functions but replaces them with an on-demand loader. Thus, even if *FaaSLight* eliminates a non-optional function by mistake, the on-demand loader can still load the code of the eliminated function when it is invoked, ensuring the correctness of the FaaS application.

Figure 3 shows the detailed architecture of *FaaSLight*, which contains two parts (i.e., **Program Analyzer** for optional function identification, and **Code Generator** for optional function elimination). More specifically, *FaaSLight* has four components (i.e., ① to ④ in the figure). Given a FaaS application, *FaaSLight* constructs the call graph to generate the final set of optional functions by using **Program Analyzer** part (Section 4.1). Then, based on the optional function set, **Code Generator** part generates the optimized FaaS application by separating optional functions from the application and designing a rewriting approach for optional functions. This rewriting approach can fetch and execute the required optional functions in an on-demand loading manner (Section 4.2).

4.1 Program Analyzer

Program Analyzer part is responsible for obtaining the final optional functions for the FaaS application. It has three components, including ① *Optional File Elimination*, ② *Application Entry Recognition*, and ③ *Optional Function Generation*. ① *Optional File Elimination* is responsible for pre-processing the given FaaS application to remove unneeded files, such as log files. Then, ② *Application Entry Recognition* find entries of the given FaaS application. ③ *Optional Function Generation* generates the final set of optional functions. Specific details of the three components are as follows.

① *Optional File Elimination.* It removes files that are not indispensable (i.e., files that are not used when the application is executed) to get a simplified FaaS application. According to the actual development process, *FaaSLight* eliminates four types of optional files for FaaS applications. (1) Files related to the virtual environment of local development. Developers may package some local files that are not related to the application functionality. For example, "pip" and "setuptools" directories may be included in the Python application; (2) the compiled files (e.g., in "pyc" or "pyi" format). These files are generated when developers test their functionalities locally, increasing the FaaS application size; (3) the information-related directories in used general libraries. For example, the "dist-info" directory only describes additional information about libraries; (4) test cases related files in used general libraries. For example, functionalities of the "tests" directory in *NumPy* library are not used by developers at all. Deleting these four types of files also can decrease the code analysis complexity of the FaaS application later.

② *Application Entry Recognition.* Generally, there are two types of functions that act as the entry of a FaaS application, including serverless functions and module initialization functions. In addition, considering the programming features of different languages, there may be language-specific functions that can play this role.

The goal of *Application Entry Recognition* component is to identify these types of entries. Next, we introduce the recognition of each type.

Serverless functions: Serverless functions are the interfaces exposed to the Internet. Since FaaS applications are executed in an event-driven fashion, the key to identifying serverless functions is to identify event handlers. We use three identification strategies. First, it is to find the configuration file (e.g., described in ".yml" or ".yaml" files), which is commonly used to configure the resource and permission of cloud applications [109]. In such a file, the relationship between serverless functions and the corresponding events can be found, further determining the name of the used serverless functions. Second, when such a configuration file does not exist in the deployment package of the FaaS application, the component analyzes the source code to find all function definitions and match the specific parameters' format. For example, on AWS Lambda, input fields of serverless functions are generally filled with "event" and "context" [27], while Google Cloud Functions-based serverless functions are filled with "request" [35].

Third, if previous strategies do not work, then the component provides an external interface to allow developers to define the name of serverless functions. Through these strategies, entry points about serverless functions can be determined for the given FaaS application. Note that there can be optional techniques for event handler identification [54, 75, 79, 114]. However, these techniques cannot be directly deployed, since they need to obtain a large number of test cases and use these cases to dynamically execute or interact with the applications, leading to too huge runtime overhead and low efficiency to be feasible in *FaaSLight*.

Module initialization functions: Module initialization functions are functions that the program needs to first call to initialize the loaded module before loading a module. *FaaSLight* detects the module initialization functions through a list generated by offline profiling. The specific process of profiling is explained as follows. We first insert probes (such as the "print" statement for the Python program and the "log" statement for the JavaScript program) to the entry of all library functions. The probes will record which functions are invoked during the runtime. Then, we load common dependency libraries (e.g., *Numpy* for Python and *request* for JavaScript) offline and get which functions are invoked during the initialization stage through the logs generated by the probes.

Language-specific functions: There are additional functions that can act as the application entry according to the characteristics of programming languages. We take Python and JavaScript, the two most widely-adopted programming languages for developing FaaS applications, as examples. For Python-based FaaS applications, magic functions are common entries [44]. They are invoked during the use of overloaded operators, and the execution process is automatically done without explicitly specifying the name of the magic function. Therefore, it is difficult to identify the calling information about such automatic execution behaviors by statically analyzing code. Since magic functions are wrapped in double underscores like "__xx__," we recognize functions in such a format as magic functions. For JavaScript-based FaaS applications, asynchronous callback functions [41] are commonly used entries. They are passed to another function as a parameter, so it is challenging to capture the function-calling information due to asynchronous handling. To tackle this problem, we analyze parameter names to match existing function definitions and then establish the relationships with callback functions. Functions that satisfy this criterion are recognized as asynchronous callback functions.

③ *Optional Function Generation.* It accepts the entries identified by ②, and outputs a set of optional functions. The idea of *Optional Function Generation* is straightforward. It traverses the call

graph and gets all functions that are reachable from the entries in the indispensable function set. Then it outputs the functions that are not in the indispensable function set as optional functions.

In our work, we perform a whole program analysis, including serverless function code and the imported third-party libraries, to build call graphs. The specific detail is as follows. We adopt a similar idea of **Class Hierarchy Analysis (CHA)** [60, 63] to identify call targets of functions. CHA is a rapid analysis technique that conservatively estimates possible calling relationships. Moreover, we use context-insensitive and flow-insensitive analyses. Such a design is enough to process serverless functions in flexible practicality and scalability. In our study, the component identifies all related called functions in the definition scope of a given function and considers them as potential call targets. However, since code features are complex and variable, the engineering implementation is not easy and may face additional challenges, e.g., alias recognition and nested functions. These challenges are solved by designing specific identification methods. For example, we analyze the parent code of the nested function to establish their mappings. Finally, the call graph is generated to capture the call relationships among functions, including different types of entries.

4.2 Code Generator

Code Generator part is the key step in optimizing the code of the FaaS application. It can achieve reduction and on-demand loading of optional code. The input of this part is the set of optional functions, and its output is to generate the optimized FaaS application. *Code Generator* part leverages the *Function-level rewriting* component to achieve its goal.

④ *Function-level Rewriting*. It is to rewrite optional functions from the FaaS application. Two main operations are accomplished: separation and rewriting, which are performed simultaneously. The separation operation separates the optional functions from the application. The workflow of the separation operation is as follows. First, for the optional function to be processed, *FaaSLight* saves the whole function definition and the body content into the corresponding value of "key-value" pairs in string format. Note that the key of "key-value" pairs is the representation of function location and function name. Then, the optional function is transferred as the corresponding function definition with an empty code body. In the rewriting operation, the empty code body is filled with our custom execution code, which has much fewer lines of code (e.g., 2 lines) than their original code. Through such a concise way, the loaded code size of the FaaS application is reduced. An example of the optional function rewriting is shown in Listing 1, whose code body has 23 lines that do not contain comments. This optional function can be transformed into 2 lines of code shown in Listing 2. Our custom execution code is to execute the "*rewrite_template*" method from the "*custom_functemplate*" module. Note that if the optional function to be processed has the parent function, and this parent function is also an optional function, the current optional function will not be rewritten, and later *FaaSLight* will directly rewrite its parent function. Such a design can reduce the number of rewritings. After handling all optional functions, the content of "key-value" pairs is generated and compressed into a global lightweight file through a compression strategy (e.g., "gzip"). Finally, all optional functions are separated out and also rewritten to finish. The lightweight file is saved in the deployment package of the FaaS application.

```
1   def load_reduce(self):
2       stack = self.stack
3       args = stack.pop()
4       func = stack[-1]
5       if len(args) and type(args[0]) is type:
6           n = args[0].__name__   # noqa
7       try:
8           stack[-1] = func(*args)
```

```
9              return
10         except TypeError as err:
11             # If we have a deprecated function ,
12             # try to replace and try again.
13             msg = "_reconstruct: First argument must be a sub-type of ndarray"
14             if msg in str(err):
15                 try:
16                     cls = args[0]
17                     stack[-1] = object.__new__(cls)
18                     return
19                 except TypeError:
20                     pass
21             elif args and issubclass(args[0], BaseOffset):
22                 # TypeError: object.__new__(Day) is not safe, use Day.__new__()
23                 cls = args[0]
24                 stack[-1] = cls.__new__(*args)
25                 return
26             raise
```

Listing 1. An example of the original function (*pandas/compat/pickle_compat/*).

```
1   def load_reduce(self):
2       import custom_functemplate
3       return custom_functemplate.rewrite_template("pandas.compat.pickle_compat
    .load_reduce", "load_reduce(self)", {"BaseOffset": BaseOffset, "self":
    self}, 1)
```

Listing 2. An example of the rewritten function.

When executing the optimized FaaS application, some optional functions are required. The "*rewrite_template*" method in the optional function first checks whether the global lightweight file is loaded. If it does not exist, then the "*rewrite_template*" method can read this file into memory and fetch the required code in the string format. If exists, then the required code can be directly fetched from memory to execute. However, the code execution of fetched optional functions may depend on some necessary variables due to the inconsistent execution context. These variables are external functionalities used in this function, and they can be generated through code analysis while rewriting the optional function.

When a FaaS application finishes all components in Figure 3, it becomes the optimized FaaS application with the necessary code.

4.3 Implementation

We have fully-fledged implemented *FaaSLight* with two execution runtime supports, one for FaaS applications written in Python and the other for ones written in JavaScript. Note that the design principles behind these two supports are inherently general according to what is presented in the preceding sections, requiring only some engineering efforts according to language-specific settings. However, these efforts are implemented inside the tool itself, while completely transparent to FaaS developers. In practice, the developer needs to only submit their FaaS application code to *FaaSLight* via the tool interface, without introducing any additional efforts. For every single FaaS application, *FaaSLight* can generate the corresponding optimized version through four components. In our approach, the main analysis builds on the foundation and improvements of CHA [60, 63], *ast* [24], *astroid* [25], and *uglify-js* [51] libraries.

5 EVALUATION

We evaluate the effectiveness of *FaaSLight* on 15 real-world FaaS applications used in Section 3. We aim to evaluate *FaaSLight* by answering the following research questions.

RQ1 (Code reduction): *How much can FaaSLight reduce the size of FaaS applications?*

RQ2 (Cold performance): *How much can FaaSLight speed up cold starts of FaaS applications?*

RQ3 (Warm performance): *How does FaaSLight affect the performance of warm starts of FaaS applications?*

RQ4 (Overhead analysis): *What is the performance overhead introduced by the on-demand loading mechanism of FaaSLight?*

RQ5 (Comparison): *How does FaaSLight perform compared with state-of-the-art methods?*

In addition, we evaluate the generalizability of our approach (i.e., loading only indispensable code) on performance optimization of FaaS applications. We aim to answer the research question about RQ6.

RQ6 (Generalizability): *Can our approach be generalized to FaaS applications written in other languages or executed on other serverless platforms?*

5.1 Evaluation Settings

We evaluate *FaaSLight* with the 15 real-world FaaS applications in Table 1 to answer the research questions of RQ1–RQ5. *FaaSLight* runs on a server with Intel Xeon 4 cores and 24 GiB main memory. The system of this server is Ubuntu 18.04.4 LTS. The tested FaaS applications are originally executed on AWS Lambda, which is the most popular and widely used serverless platform [20, 21]. In our study, original FaaS applications are denoted as *before* applications. FaaS applications processed by *Optional File Elimination* component of *FaaSLight* are denoted as *after1* applications, and the final optimized FaaS applications are denoted as *after2* applications.

We run experiments on each of the 15 tested FaaS applications 20 times to collect performance metrics (e.g., latency and memory usage) and use *Mann Whitney U-test* [42, 89] (which is suitable for the small sample size and does not require normality) to measure the statistical significance. When comparing two sets of performance results for *after2* and *before* applications, the null hypothesis is that the performance of *after2* set is similar to *before* set. The threshold of statistical significance is set as p-value < 0.05. We further compute the effect size as the Cohen's d [33], to check if the difference has a meaningful effect. d is between 0 and 2, where 0.2 indicates a small effect, 0.5 a medium effect, and 0.8 a large effect [33].

5.2 RQ1: Code Reduction

To explore how *FaaSLight* reduces the application code size and hence code loading time, we compare the application package size (*Size*), the number of functions (*FC*), and lines of code (*LoC*) for *before*, *after1*, and *after2* applications, respectively. The percentage of these values for *after1* and *after2* applications compared to the *before* application is shown in Figure 4. Specifically, in the best case, *Size* of *after2* applications becomes 60.84% of *before* applications, reducing 39.16% optional content. On average, *FaaSLight* makes the size of *before* applications decrease to their 84.82%, reducing 15.18% *Size*. For *FC* and *LoC* of *after2* applications, *FaaSLight* reduces by 55.06% and 58.75% of the original *FC* and *LoC*, respectively, on average.

From Figure 4, we also find that the reduction of statistical values is mainly in *after1* applications, which means that *Optional File Elimination* component can effectively decrease a large part of optional files. Based on *after1* versions, *FaaSLight* leverages *Code Generator* part to merge and simplify optional functions that are loaded, i.e., rewriting optional functions, to further decrease statistical values to get the *after2* version. In addition, when executing the optimized FaaS
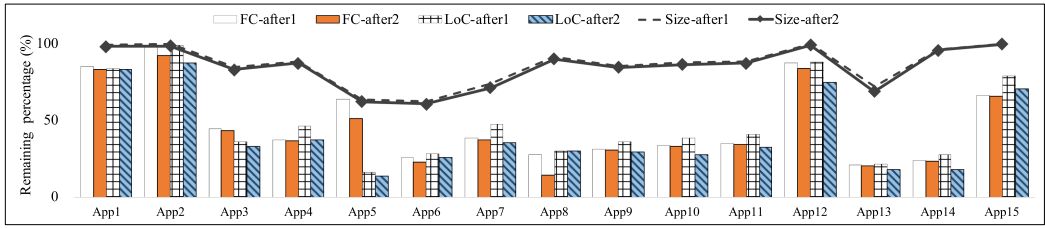
Fig. 4. The change of statistical values (the optimized application as a percentage of the original application).

applications, on average, *FaaSLight* callbacks only 10 optional functions according to given input cases on demand.

> **Answer to RQ1:** On average, *FaaSLight* reduces the application size by 15.18%, the number of functions by 55.06%, and the number of code lines by 58.75%.

5.3 RQ2: Cold Performance

To answer how *FaaSLight* speeds up the performance in cold starts, we compare the preparation phase latency, loading phase latency, total response latency, and runtime memory for *before*, *after1*, and *after2* applications, respectively. The specific explanation of these latencies can be found in Section 3.2. In our approach, the performance improvement of FaaS applications is mainly attributed to two components. The first one is the *Optional File Elimination* component, which reduces optional files. The second one is the *Function-level Rewriting* component, which separates optional code from the loaded application code. *Optional File Elimination* component can reduce the application size, thus decreasing the application transmission time, i.e., the preparation phase latency. *Function-level Rewriting* component can directly decrease the loading code size for FaaS applications, thus reducing the loading phase latency. Table 2 shows the real latency results for tested FaaS applications. Meanwhile, Figures 5, 6, and 7 show reduction percentages of the preparation phase latency, loading phase latency, and total response latency for *after1* and *after2* applications in cold starts, respectively. Specifically, as shown in Figure 5, *FaaSLight* reduces 3.40% to 26.14% of the preparation phase latency except for App5, App12, and App15. The potential reasons are explained as follows. The application size of App5 is small (i.e., 25.26 MB), so the reduction may not be enough to influence the application transmission latency in the preparation phase. Application code sizes of App12 and App15 are large, but their optional files are few, since the application consists mainly of dependency libraries with few local development files. Therefore, the preparation phase latencies of App5, App12, and App15 do not change much. *FaaSLight* reduces the preparation phase latency by up to 26.14% (11.72% on average).

For the loading phase latency, *Code Generator* part is an effective way to decrease this latency by rewriting optionally loaded functions as ones with only two lines of code. Specifically, *FaaSLight* makes the loading phase latency reduced by up to 78.95%. On average, FaaS applications have 28.78% performance improvement on the loading phase latency. Especially for App2, App7, and App8, *FaaSLight* can reduce more than 60% loading phase latency due to the simplicity of the tasks.

For the total response latency, the reduction percentage for *after1* and *after2* applications is shown in Figure 7. Specifically, for the final *after2* applications, *FaaSLight* makes the total response latency reduced by up to 42.05% (19.21% on average). However, we also observe that some FaaS applications, e.g., App1, App5, and App12, obtain a small performance improvement. The potential reasons are explained as follows. First, these FaaS applications have a low improvement in the preparation phase latency, as previously mentioned. Second, since the functionalities of

Table 2. Performance Results of Python-based FaaS Applications in Cold Starts on AWS Lambda

App ID	Version	Preparation phase latency (ms)	Loading phase latency (ms)	Runtime memory (MB)	Total response latency (ms)
App1	before	1,420.43	760.95	93	4,011.32
	after1	1,279.10 (**−9.95%**)	748.53	93	3,792.86 (**−5.45%**)
	after2	1,270.38 (**−10.56%**)	701.82 (**−7.77%**)	90 (**−3.23%**)	3,565.94 (**−11.10%**)
App2	before	1,463.63	873.00	68	2,468.94
	after1	1,296.08 (**−11.45%**)	870.86	68	2,316.69 (**−6.17%**)
	after2	1,279.88 (**−12.55%**)	183.77 (**−78.95%**)	43 (**−36.76%**)	1,602.61 (**−35.09%**)
App3	before	1,532.66	1,794.47	115	3,522.86
	after1	1,343.44 (**−12.35%**)	1,782.62	115	3,323.86 (**−5.65%**)
	after2	1,232.81 (**−19.56%**)	1,708.84 (**−4.77%**)	113 (**−1.74%**)	3,131.91 (**−11.10%**)
App4	before	2,011.29	2,352.07	142	4,593.43
	after1	1,778.86 (**−11.56%**)	2,235.33 (**−4.96%**)	141	4,163.48 (**−9.36%**)
	after2	1,768.07 (**−12.09%**)	2,033.82 (**−13.53%**)	140 (**−1.41%**)	4,004.10 (**−12.83%**)
App5	before	1,279.98	488.81	62	2,689.65
	after1	1,249.72	451.97 (**−7.54%**)	61	2,601.61 (**−3.27%**)
	after2	1,272.52	435.84 (**−10.84%**)	60 (**−3.23%**)	2,511.24 (**−6.63%**)
App6	before	1,540.03	1,959.92	125	3,660.67
	after1	1,398.48 (**−9.19%**)	1,774.72 (**−9.45%**)	115 (**−8.00%**)	3,346.29 (**−8.59%**)
	after2	1,340.51 (**−12.96%**)	1,536.38 (**−21.61%**)	107 (**−14.40%**)	3,054.51 (**−16.56%**)
App7	before	2,312.04	4,580.31	228	7,165.54
	after1	2,185.66 (**−5.47%**)	4,217.68 (**−7.92%**)	206 (**−9.65%**)	6,770.75 (**−5.51%**)
	after2	2,177.49 (**−5.82%**)	1,408.62 (**−69.27%**)	130 (**−42.98%**)	4,152.73 (**−42.05%**)
App8	before	1,739.22	887.81	102	2,741.02
	after1	1,642.62 (**−5.55%**)	790.91 (**−10.92%**)	98 (**−3.92%**)	2,562.48 (**−6.51%**)
	after2	1,620.82 (**−6.81%**)	188.26 (**−78.80%**)	42 (**−58.82%**)	1,951.16 (**−28.82%**)
App9 train	before	2,741.06	3,985.42	230	9,035.39
	after1	2,140.74 (**−21.90%**)	3,790.63 (**−4.89%**)	229	8,218.25 (**−9.04%**)
	after2	2,108.48 (**−23.08%**)	3,135.82 (**−21.32%**)	216 (**−6.09%**)	7,470.49 (**−17.32%**)
App9 predict	before	2,700.32	3,828.55	230	8,291.80
	after1	2,188.82 (**−18.94%**)	3,689.76 (**−3.63%**)	229	7,712.55 (**−6.99%**)
	after2	1,994.47 (**−26.14%**)	3,141.81 (**−17.94%**)	215 (**−6.09%**)	7,071.03 (**−14.72%**)
App10	before	2,365.90	2,391.77	159	4,961.16
	after1	2,081.80 (**−12.01%**)	2,272.51 (**−4.99%**)	158	4,494.80 (**−9.40%**)
	after2	1,914.79 (**−19.07%**)	1,895.94 (**−20.73%**)	148 (**−6.92%**)	4,035.48 (**−18.66%**)
App11	before	2,018.32	3,384.63	182	5,551.03
	after1	1,943.81 (**−3.69%**)	3,308.98 (**−2.23%**)	181	5,407.95 (**−2.58%**)
	after2	1,949.72 (**−3.40%**)	1,722.93 (**−49.10%**)	141 (**−22.53%**)	3,934.31 (**−29.12%**)
App12 (Docker)	before	1,266.72	6,966.72	410	8,442.14
	after1	1,255.25	6,901.35	410	8,281.06 (**−1.91%**)
	after2	1,279.19	6,036.30 (**−13.36%**)	397 (**−3.17%**)	7,448.55 (**−11.77%**)
App13	before	1,441.96	701.41	78	2,304.29
	after1	1,252.35 (**−13.15%**)	646.34 (**−7.85%**)	74 (**−5.13%**)	2,052.75 (**−10.92%**)
	after2	1,241.88 (**−13.88**)	571.64 (**−18.50%**)	69 (**−11.54%**)	1,961.71 (**−14.87%**)
App14	before	2,592.21	953.36	116	3,678.09
	after1	2,153.25 (**−16.93%**)	938.65 (**−1.54%**)	116	3,195.03 (**−13.13%**)
	after2	2,032.94 (**−21.58%**)	830.00 (**−12.94%**)	114 (**−1.72%**)	2,980.90 (**−18.96%**)
App15 (Docker)	before	1,368.85	6,211.63	872	8,062.10
	after1	1,319.33	5,923.85 (**−4.63%**)	872	7,840.81 (**−2.74%**)
	after2	1,307.50	4,902.64 (**−21.07%**)	732 (**−16.06%**)	6,635.88 (**−17.69%**)
Max		**26.14%**	**78.95%**	**58.82%**	**42.05%**
Mean		**11.72%**	**28.78%**	**14.79%**	**19.21%**

dependency libraries are well written, the optional code that can be separated from the FaaS application is small, which makes the reduction in the loading phase latencies limited. To explore the performance improvement effect, we calculate *Mann Whitney U-test* of all measurements about the total response latency between *after2* and *before* applications. Results are shown in Table 3, where "*" represents that the p-value is less than 0.05. We observe that 14 (14/15 = 93.33%) optimized FaaS applications have a statistically different performance from their original FaaS applications. In these FaaS applications, eight tested FaaS applications (App1, App2, App4, App6, App7, App8,
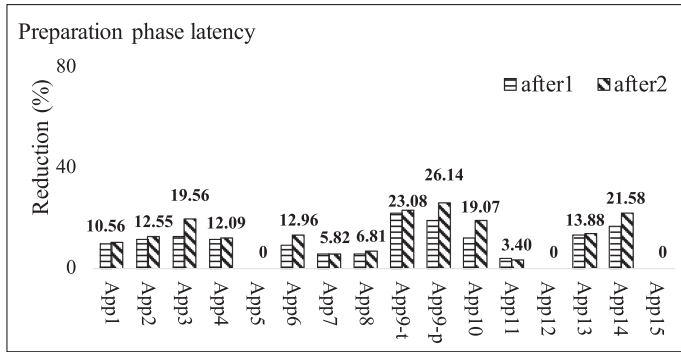
Fig. 5. The reduction percentage of the preparation phase latency for *after1* and *after2* applications in cold starts.
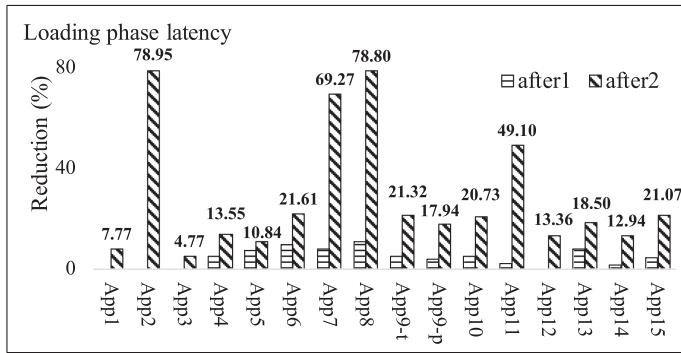
Fig. 6. The reduction percentage of the loading phase latency for *after1* and *after2* applications in cold starts.
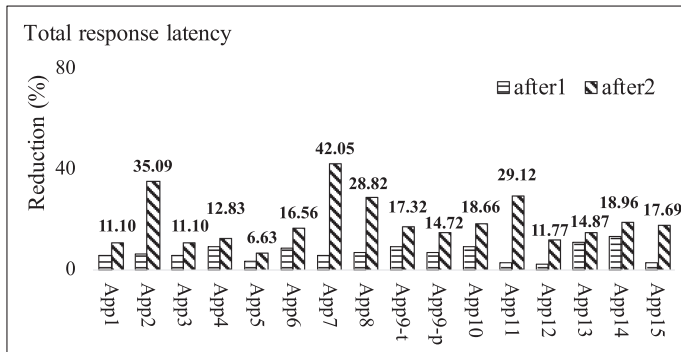
Fig. 7. The reduction percentage of the total response latency for *after1* and *after2* applications in cold starts.

App9-t, App9-p, and App11) show large effect sizes (>= 0.8), i.e., large performance improvement effect. Five tested FaaS applications (App3, App10, App13, App14, and App15) show medium effect sizes (>= 0.5), i.e., medium performance improvement effect. Moreover, the effect sizes of these five FaaS applications are nearly 0.8, indicating that they have a relatively large performance improvement effect at the medium effect level. Only one tested FaaS application (App5) shows a small effect size (>= 0.2), i.e., a small performance improvement effect. However, its effect size is 0.48, which is

Table 3. Statistical Test Results of the Total Response Latency between *after2* and *before*
Applications in Cold Starts

	App1	App2	App3	App4	App5	App6	App7	App8
Effect size	0.81*	0.96*	0.74*	0.80*	0.48*	0.84*	0.99*	0.87*
	App9-t	App9-p	App10	App11	App12	App13	App14	App15
Effect size	0.85*	0.80*	0.79*	0.98*	—	0.79*	0.78*	0.75*

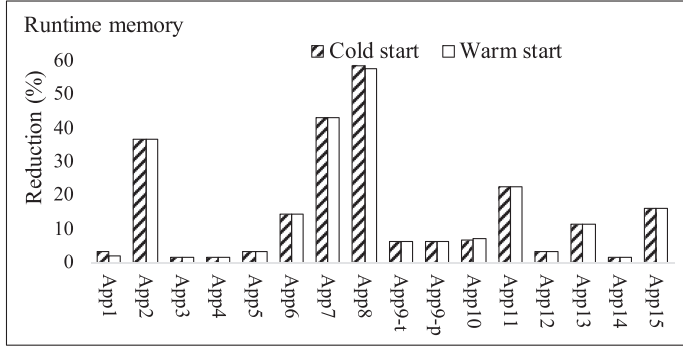The symbol "*" represents that the p-value is less than 0.05.

Fig. 8. The reduction percentage of the runtime memory for the final *after2* applications in cold and warm starts.

nearly 0.5. Similarly, App5 shows a relatively large performance improvement effect at the small effect level. Overall, *FaaSLight* can significantly improve the total performance of FaaS applications.

As shown in Figure 4, *after1* applications that apply *Optional File Elimination* component of *FaaSLight* can reduce more optional files. However, these optional files are mostly from files unrelated to the application loading code. Directly deleting such files helps FaaS applications decrease the application transmission latency in the preparation phase. The application transmission latency accounts for a small percentage of overhead in the cold-start latency. Therefore, the improvement effect of *after1* applications is limited, shown in Figures 6 and 7. *Program Analyzer* part identifies optional functions, which are major consumers of the application code loading latency. Therefore, as shown in Figures 6 and 7, the performance improvement of *after2* applications is more effective than that of *after1* applications.

FaaSLight makes the runtime memory reduce by up to 58.82% (on average 14.79%) in cold starts, shown in Figure 8. In addition, in our study, some tested FaaS applications (e.g., App12 and App15) have a "big" application code size, exceeding the normal deployment size limit. They are deployed by the container image.

The billed duration of this deployment way is the sum of the function execution latency and application code loading latency. Thus, reducing the loading phase latency is beneficial to reduce the developer's billed duration. Results show that *FaaSLight* reduces 13.34% to 20.71% billed duration of heavy FaaS applications, such as App12 and App15.

Answer to RQ2: *FaaSLight* reduces the preparation phase latency by up to 26.14% (on average 11.72%), application code loading latency by up to 78.95% (on average 28.78%), total response latency by up to 42.05% (on average 19.21%). Moreover, the performance improvement achieved by *FaaSLight* is statistically significant for 93.33% of the studied FaaS applications. As an additional benefit, *FaaSLight* decreases the runtime memory by up to 58.82% (on average 14.79%).

5.4 RQ3: Warm Performance

To explore the effect of *FaaSLight* on warm starts of FaaS applications, we compare their scheduling phase latency, total response latency, and runtime memory. Results show that *FaaSLight* does not increase the scheduling phase latency and total response latency, meaning that the performance is maintained in the original warm execution performance. We also calculate *Mann Whitney U-test* for all measurements about the total response latency between *after2* and *before* applications. The p-value is all large than 0.05, indicating that optimized applications do not have statistically different performances than original ones in warm starts. We further explore the runtime memory and find that *FaaSLight* also reduces the runtime memory by up to 57.84% (on average 14.74%) in warm starts shown in Figure 8. It guides developers to configure lower billing memory.

> **Answer to RQ3:** *FaaSLight* has no observable effect on the performance of FaaS applications in warm starts and reduces the runtime memory by up to 57.84% (on average 14.74%).

5.5 RQ4: Overhead Analysis

FaaSLight adopts an on-demand loading strategy that fetches optional functions when they are required. This strategy may potentially increase the function execution latency and, therefore, increase its warm-start latency. In Section 5.4, we confirm that *FaaSLight* does not introduce observable latency to warm starts. This section further studies how *FaaSLight* causes runtime overhead and why it does not cause observable delays to warm starts of the FaaS applications.

We measure the runtime cost introduced by the on-demand loading strategy to answer this research question. We find that the latency overhead of this strategy is about 100 ms, on average, due to reading the lightweight file that saves separated optional functions. Certainly, the reading latency is affected by this file size. When this file saves about 5,000 optional functions, its size is only about 1 MB due to our content compression strategy. Its reading latency is between 110 ms to 150 ms.

The runtime cost of the on-demand loading strategy does not affect the warm-start latency of FaaS applications, because it is a one-time cost if the container or VM of a serverless function is not released. Specifically, when the first optional function in invoked, *FaaSLight* reads the lightweight file and loads all optional functions into the memory of the container or VM. As long as the serverless platform does not recycle the function execution environment, optional functions already loaded can be used to assist in subsequent executions of the serverless function. Assume a container instance of a serverless function serves ten requests before it is released. In this case, on-demand loading only happens to the request that first invokes an optional function. This request produces the additional runtime cost, which is added to the function execution latency. However, the other nine requests will not be affected, since the memory has loaded all optional functions. Note that the ten requests consist of one cold start and nine warm starts. Although the overhead (100 ms) of the on-demand loading strategy increases the function execution latency, *FaaSLight* has made the application code loading latency reduce by 28.78% on average (about 1,000 ms). Therefore, it is beneficial to trade the one-time 100 ms execution latency for the reduction of the cold-start latency. Moreover, it is worth sacrificing a little execution performance overhead for greater overall performance optimization (on average, 19.21%) of FaaS applications.

> **Answer to RQ4:** The on-demand loading strategy of *FaaSLight* introduces a small runtime cost (about 100 ms).
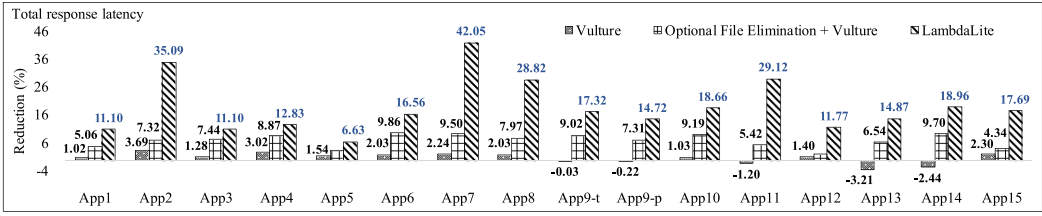
Fig. 9. The reduction percentage of the total response latency for different methods in cold starts.

5.6 RQ5: Comparison

To further demonstrate the effectiveness of *FaaSLight*, we compare it with a well-known program analysis tool called *Vulture* [22], which can also identify optional code for Python applications and has been widely adopted in the industry [19, 39].

Vulture identifies the objects that have been defined but not used in all given Python files, and reports them as optional code.

Vulture and *FaaSLight* are both not related to input cases, focusing on statically analyzing the application code. In our study, we apply *Vulture* to our benchmarks to obtain optional functions. Moreover, we also use a mixed method that combines the functionality of *Optional File Elimination* component and *Vulture*. Optional functions identified by *Vulture* are separated and rewritten by *Code Generator* part of *FaaSLight*, to be able to load them on demand to ensure the availability of the optimized FaaS application. The results of these methods are shown in Figure 9.

We first compare the performance improvement of *Vulture* method and *FaaSLight*. On average, *Vulture* method shows 0.90% performance improvement on the total response latency, while *FaaSLight* can achieve 19.21% improvement. It illustrates that the latency improvement of *FaaSLight* is 21.25× that of *Vulture*. In addition, in the best case, *Vulture* method obtains 3.69% performance improvement, while *FaaSLight* achieves 42.05% improvement. Moreover, *Vulture* method does not achieve performance improvement on some FaaS applications, such as App9, App11, App13, and App14. The reason is that the number of optional functions verified by *Vulture* is small (only 400 on average). When the optimized FaaS application needs some optional functions to trigger the on-demand loading, the latency improvement brought by separating optional functions from the application is not enough to compensate for the overhead of reading the file of optional functions. On the contrary, *FaaSLight* removes some optional files through *Optional File Elimination* component, and separates on average 5,000 optional functions that are loaded through *Code generator* part. Thus, *FaaSLight* shows the effective performance improvement in all tested FaaS applications.

We further analyze the design principle of *Vulture*. It identifies only the function objects that have been defined but not used in code. Such an analysis lacks a global overview of function usage related to application functionalities. Some functions may be both defined and used in code, but they may be optional for application functionalities. In this situation, *Vulture* misses many functions that may be optional, making the number of separated optional functions small. Therefore, *Vulture* is not effective enough to optimize the total response latency of FaaS applications. On the contrary, *FaaSLight* can identify the relevant reachable functions starting from entry points, i.e., serverless functions that represent application functionalities. Functions not related to application functionalities are viewed as optional functions and separated from the FaaS applications.

We also compare the impact of critical parts on the improvement of the total response latency. Compared with the *Vulture* method and the mixed method, the effect of our *Optional File*

Elimination component can be analyzed. The mixed method obtains a 7.09% improvement in reducing the total response latency on average. It illustrates that our *Optional File Elimination* component has a positive impact on performance improvement, speeding up *Vulture* 7.85×. Compared with the mixed method and *FaaSLight*, the ability to identify optional functions of loaded code can be analyzed for *Vulture* and our key *Program Analyzer* part. Results show that *FaaSLight* improves by 12.12% on the basis of the mixed method. In this situation, the reduction of the total response latency of *FaaSLight* is 2.71× that of the mixed method. It illustrates that our *Program Analyzer* part is stronger than *Vulture* on the effectiveness of the optional function identification.

To sum up, critical parts of *FaaSLight* have a positive impact on optimizing the total response latency of serverless functions.

> **Answer to RQ5:** Compared with the state-of-the-art, *FaaSLight* achieves a 21.25× improvement in reducing the average total response latency.

5.7 RQ6: Generalizability

For the performance optimization of FaaS applications, we adopt the approach of loading only indispensable code for FaaS applications. To demonstrate the generalizability of the proposed approach, we compare the performance of FaaS applications written in different programming languages (i.e., Python and JavaScript) and executed on different serverless platforms (i.e., AWS Lambda and Google Cloud Functions) before and after optimization, respectively.

5.7.1 Language Generalizability. To demonstrate the language generalizability of *FaaSLight*, we further evaluate it on FaaS functions written in JavaScript, which as well as Python are two of the most widely used and popular programming languages in the serverless community [5, 50, 65, 66]. To this end, we implement our approach as the JavaScript prototype based on call graph analysis [31, 56], our customization of program analysis, and optimization of on-demand loading strategy. The source code has been released in our GitHub repository.

We select the JavaScript-based FaaS applications from the "Wonderless" dataset [66]. This dataset collects applications developed in a serverless fashion from GitHub, and these applications are mainly executed on AWS Lambda. Considering the functionality executability of FaaS applications and the scale representativeness of their application size, finally, we select four FaaS applications written in JavaScript and executed on AWS Lambda. Table 4 shows their specific information, including the application size (*Size*) and the number of lines of code (*LoC*). These FaaS applications are processed by our approach. We compare the related latencies of *before* and *after2* applications in cold starts. The result in Table 4 shows that the preparation phase latency, loading phase latency, and total response latency of *after2* applications are all lower compared with *before* applications. Specifically, the preparation phase latency decreases by 4.27% to 11.41%, and our approach reduces the loading phase latency by up to 18.38%. Finally, the optimized FaaS applications obtain 8.34% to 10.44% performance improvement in the total response latency. Overall, these results show that loading only indispensable code can improve the performance of FaaS applications written in JavaScript.

5.7.2 Platform Generalizability. To demonstrate the platform generalizability of our approach, we use Google Cloud Functions, another widely adopted serverless platform [20, 21, 50, 64, 68], for evaluation.

First, we evaluate *FaaSLight* for the aforementioned FaaS applications, i.e., 15 Python applications (i.e., App1 to App15) and 4 JavaScript applications (i.e., App16 to App19), on Google Cloud Functions. These applications were developed for AWS Lambda, and cannot be directly migrated

Table 4. Performance Result of FaaS Applications Written in JavaScript and Executed in AWS Lambda (Cold Starts)

App ID	Name	Version	Size (MB)	LoC (k)	Preparation phase latency (ms)	Loading phase latency (ms)	Total response latency (ms)
App16	lambda-request [43]	before	4.29	51.32	1,377.12	355.44	2,368.31
		after2	3.38	30.80	1,290.61 (−**6.28%**)	303.69 (−**14.56%**)	2,137.97 (−**9.73%**)
App17	handlebars-fetch [49]	before	10.21	125.91	1,195.14	229.29	1,892.45
		after2	6.67	954.19	1,058.72 (−**11.41%**)	187.15 (−**18.38%**)	1,694.88 (−**10.44%**)
App18	request-cheerio [47]	before	13.03	161.42	1,177.60	457.41	1,853.85
		after2	11.36	111.21	1,084.40 (−**7.91%**)	404.42 (−**11.59%**)	1,699.21 (−**8.34%**)
App19	serverless-image [46]	before	101.95	499.43	970.98	744.49	2,088.38
		after2	97.21	383.52	929.47 (−**4.27%**)	677.19 (−**9.04%**)	1,907.12 (−**8.68%**)

to and executed on Google Cloud Functions without additional engineering efforts. This is because different serverless platforms may have different requirements for executing the applications. For example, AWS Lambda and Google Cloud Functions have different definition formats of serverless functions [27, 35].

To tackle this problem, we manually convert these applications for AWS Lambda to meet the requirements of Google Cloud Functions. To alleviate the potential threat in the implementation of application conversion and migration, we have invited a professional developer, who has five-year practice and experience in developing industry-level FaaS applications, to help review and test our modified code and ensure their correctness. We find that App12 and App15 cannot be migrated and executed, since they exceed the 500 MB limit of the deployment package size that is supported by Google Cloud Functions [45]. Therefore, we use the remaining 13 applications for evaluation. All the source code of these applications along with the instructions on how we modify the code, have been released on our GitHub.

Table 5 shows the performance of these applications on Google Cloud Functions. The result shows that Google Cloud Functions has a similar cold-start problem as AWS Lambda, i.e., the cold-start latency contains the non-negligible application code loading overhead. Thus, reducing loaded code size can help to optimize the application performance. Specifically, on Google Cloud Functions, our approach can reduce the total response latency of serverless functions by up to 32.20%, with the help of 2.92% to 14.77% of preparation phase latency optimization and 8.87% to 77.15% of loading phase latency optimization. Overall, these results show that loading only indispensable code can improve the performance of FaaS applications executed on Google Cloud Functions.

To further evaluate the platform generalization, we also collect real-world FaaS applications that were originally designed and executed on Google Cloud Functions. We mine such applications from GitHub using the "Google Cloud Functions" keyword. Considering the functionality executability of FaaS applications and the scale representativeness of their application size, we finally select three new FaaS applications written in Python. We show the details of these applications (i.e., App20, App21, and App22) in Table 6. We evaluate *FaaSLight* through these applications on both Google Cloud Functions and AWS Lambda. To this end, we follow the previous steps to convert these applications into the format required by AWS Lambda. Table 7 presents the performance results of these applications. The results show that the effectiveness of *FaaSLight* holds for these applications on both Google Could Functions and AWS Lambda, further demonstrating the platform generalization of *FaaSLight*. Specifically, on Google Could Functions, *FaaSLight* reduces around 12.62% of the total response latency for these applications; on AWS Lambda, *FaaSLight* reduces around 11.37% of the total response latency. The different performance for different platforms can be explained by the findings in previous studies [80, 107, 110] that different platforms can exhibit different performance results even for the same application.

Table 5. Cold-start Result of App1 to App19 (Originally Developed for AWS Lambda)
on Google Cloud Functions

App ID	Version	Preparation phase latency (ms)	Loading phase latency (ms)	Total response latency (ms)
App1	before	3,090.93	1,410.13	6,006.80
	after2	2,634.28 (−**14.77%**)	1,258.78 (−**10.73%**)	5,252.11 (−**12.56%**)
App2	before	2,357.36	1,172.75	3,859.45
	after2	2,200.20 (−**6.67%**)	267.97 (−**77.15%**)	2,764.36 (−**28.37%**)
App3	before	2,679.33	2,921.13	5,957.00
	after2	2,348.42 (−**12.35%**)	2,594.37 (−**11.19%**)	5,312.04 (−**10.83%**)
App4	before	2,546.22	3,990.18	6,923.60
	after2	2,389.02 (−**6.17%**)	3,503.52 (−**12.20%**)	6,222.04 (−**10.13%**)
App5	before	2,520.70	667.05	5,062.62
	after2	2,447.07 (−**2.92%**)	607.86 (−**8.87%**)	4,519.87 (−**10.72%**)
App6	before	2,664.78	2,925.26	5,904.28
	after2	2,351.88 (−**11.74%**)	2,417.19 (−**17.37%**)	5,124.88 (−**13.20%**)
App7	before	2,700.94	4,444.32	7,818.41
	after2	2,490.25 (−**7.80%**)	2,203.49 (−**50.42%**)	5,301.07 (−**32.20%**)
App8	before	2,727.69	1,289.20	4,369.73
	after2	2,548.21 (−**6.58%**)	326.21 (−**74.70%**)	3,187.51 (−**27.05%**)
App9 train	before	2,784.74	7,771.63	12,687.07
	after2	2,498.84 (−**10.27%**)	6,480.68 (−**16.61%**)	10,863.83 (−**14.37%**)
App9 predict	before	2,756.00	7,880.20	12,578.81
	after2	2,485.14 (−**9.83%**)	6,638.54 (−**15.76%**)	11,175.23 (−**11.16%**)
App10	before	2,697.58	4,201.91	7,471.82
	after2	2,505.34 (−**7.13%**)	3,683.43 (−**12.34%**)	6,573.01 (−**12.03%**)
App11	before	2,688.60	7,127.09	10,245.73
	after2	2,420.26 (−**9.98%**)	5,201.01 (−**27.02%**)	7,945.03 (−**22.46%**)
App13	before	2,928.96	915.75	4,181.88
	after2	2,632.38 (−**10.13%**)	798.41 (−**12.81%**)	3,735.14 (−**10.68%**)
App14	before	2,767.26	1,213.27	4,285.15
	after2	2,369.78 (−**14.36%**)	966.29 (−**20.36%**)	3,678.36 (−**14.16%**)
App16	before	1,905.50	504.45	2,748.74
	after2	1,753.29 (−**7.99%**)	383.82 (−**23.91%**)	2,457.23 (−**10.61%**)
App17	before	2,063.34	202.69	2,629.53
	after2	1,840.20 (−**10.81%**)	173.23 (−**14.54%**)	2,376.29 (−**9.63%**)
App18	before	1,959.28	591.09	3,096.21
	after2	1,846.48 (−**5.76%**)	504.00 (−**14.73%**)	2,865.85 (−**7.44%**)
App19	before	1,826.22	646.50	2,795.05
	after2	1,687.23 (−**7.61%**)	556.60 (−**13.91%**)	2,580.41 (−**7.68%**)

App12 and App15 are excluded because they exceed the 500 MB uncompressed size limit of Google Cloud Functions.

> **Answer to RQ6:** Our approach can be generalized to performance optimization of FaaS applications written in JavaScript languages (another widely used language in serverless computing) or executed on Google Cloud Functions (another popular serverless platform). The application code loading latency is reduced by up to 77.15%, thus decreasing up to 32.20% of the total response latency. The results have evidenced that our approach can be potentially generalized to a variety of FaaS applications with heterogeneous implementation and underlying serverless platforms.

6 THREATS TO VALIDITY

Internal validity. In our measurement study, we explore the possible root cause of the cold-start overhead of FaaS applications. Since FaaS applications may be affected by resource allocation or the network of the serverless platform, the obtained latencies may lead to possible percentage bias in Figure 2. To alleviate this threat, it has been a common practice in the serverless computing literature to calculate the average latency of multiple runs to represent the general level

Table 6. Details of the Newly Added FaaS Applications
Executed on Google Cloud Functions

App ID	Name	Version	Size (MB)	LoC (k)
App20	google-requests [37]	before	6.09	41.75
		after2	2.94	25.56
App21	google-pandas [36]	before	122.80	420.93
		after2	82.22	196.24
App22	LDA-classifier [32]	before	337.53	606.34
		after2	316.35	427.27

Table 7. Cold-start Result* of App20 to App22 (Originally Developed for Google Cloud Functions) on
Both AWS Lambda and Google Cloud Functions

App ID	Executed platform	Version	Preparation phase latency	Loading phase latency	Total response latency
App20	AWS Lambda	before	900.06	708.70	2,082.54
		after2	848.48 (−**5.73 %**)	608.90 (−**14.08%**)	1,890.49 (−**9.22%**)
	Google Cloud Functions	before	2,146.06	345.18	3,308.11
		after2	2,025.58 (−**5.61%**)	305.85 (−**11.39%**)	2,972.61 (−**10.14%**)
App21	AWS Lambda	before	1,149.46	3,585.32	5,137.01
		after2	1,007.52 (−**12.35%**)	2,972.49 (−**17.09%**)	4,378.16 (−**14.77%**)
	Google Cloud Functions	before	2,476.64	4,194.23	7,011.91
		after2	2,112.53 (−**14.70%**)	3,306.28 (−**21.17%**)	5,731.38 (−**18.26%**)
App22	AWS Lambda	before	1,064.06	1,677.95	2,834.28
		after2	981.71 (−**7.74%**)	1,459.05 (−**13.05%**)	2,547.58 (−**10.12%**)
	Google Cloud Functions	before	2,243.30	2,912.45	5,972.66
		after2	2,097.86 (−**6.48%**)	2,562.56 (−**12.01%**)	5,408.63 (−**9.44%**)

* Performance results are in milliseconds.

[80, 107, 111]. In our study, we follow the previous studies [80, 107, 111] to run the experiments multiple times to report the average results.

For our measurement study and the experimental evaluation, we measure FaaS applications 20 times and then use the average value as the final comparable result of the performance.

In addition, in our study, we identify indispensable functions of FaaS applications by constructing the function-level call graph. The inaccuracy or incompleteness of the call graph may lead to missing some indispensable functions to cause application failure. To mitigate the threat, *FaaSLight* adopts the strategy of identifying as many indispensable functions as possible. Moreover, we design a mechanism for FaaS applications to fetch and execute optional functions in an on-demand loading way. In future work, we plan to design a more accurate code identification for FaaS applications while guaranteeing the correctness and effectiveness of FaaS applications.

External validity.

We evaluate *FaaSLight* with real-world Python-based FaaS applications executing on AWS Lambda. This may lead to the limited generalizability of *FaaSLight* to FaaS applications written in other languages or executed on other serverless platforms. To mitigate this threat, we present and answer the research question, i.e., RQ6. The result shows that our approach can be generalized to performance optimization of FaaS applications written in JavaScript or executed on Google Cloud Functions, indicating the generalizability of our approach. For compiled languages like C/C++, although optimizations are already done at compile time, used standard libraries are also imported into applications [74, 93]. Similar to FaaS applications, applications may bear the performance burden of carrying all the features in the code with no way to disable or remove optional features. Reducing unnecessary feature-related code in compile phase means code size reduction, which is helpful for performance improvement of the program written in compiled languages [59, 91, 98]. Therefore, loading only indispensable code can also be applicable to performance improvement of FaaS applications written in compiled languages.

7 RELATED WORK

Serverless computing has been used in a wide range of software applications [57, 90, 97], and thus attracted increasing attention from the SE community [17, 65, 95, 108–110]. Some measurement studies [17, 110] have been presented to characterize the performance of serverless applications or platforms to help developers select the most appropriate one. To facilitate developers to develop their serverless applications, Wen et al. [109] uncovered 36 specific challenges that developers encounter in developing serverless applications. Performance-related challenges are also included. A comprehensive study about serverless applications [65] was presented by SE researchers to show specific usage characteristics, where performance is also a reason for serverless adoption. In addition to the attention from the SE community, other communities like the Systems research community, the Networks research community, and the Services Computing research community have made significant efforts related to serverless computing in performance optimization [55, 70, 85], resource management [87, 112], communication optimization [76, 99], and so on. In our study, we present an application-level code analysis approach to optimize the code of FaaS applications. This approach can be adopted by software developers to improve the cold-start performance of FaaS applications.

According to the related studies [52, 85, 104], there are two types of approaches to cold-start optimization in serverless computing. The first one is to reduce the number of cold starts through a "keep-alive" or pre-warm policy. For example, major serverless platforms like AWS and Azure use a fixed "keep-alive" policy to retain the resources in memory for several minutes after a function execution [52, 53]. Although such a policy is simple and practical, it does not consider the actual invocation frequency and function patterns. Therefore, there are still many cold starts for most requests. Moreover, developers can easily identify this policy, causing them to keep resources warm by making frequent dummy invocations in advance. This practice exacerbates the resource waste problem. Another approach is to present new systems by designing data caching mechanisms [85], managing container runtime [55], or setting and restoring snapshots [64, 100] in the underlying platforms. Specifically, SOCK [85] cached interpreters and commonly used libraries in containers to reduce the cold-start latency, and they provided the lightweight isolation mechanism for serverless functions. SAND [55] applied the application-level sandbox runtime sharing to reduce the number of containers and, thus, container preparation latency. Silva et al. [100] proposed the usage of checkpoint/restore. The designed system prototype can restore snapshots of previously started functions' runtime instead of re-executing new cold starts. Similarly, Catalyzer [64] set checkpoints of critical paths and restored application and sandbox runtime in Google's gVisor [38]. However, these optimization studies have modified underlying platform designs or sandbox runtime mechanisms; thus, it is difficult to apply in presented infrastructures on different platforms due to extensive engineering efforts, maintenance, and security problems. Differently, our approach effectively optimizes the cold-start latency at the application level, and it allows developers to improve the performance of their FaaS applications without any additional overhead.

8 DISCUSSION

Accurate detection and optional code backup. In our article, we use static program analysis to construct the call graph, because dynamic analysis generally produces huge runtime overhead, which is inflexible and impractical in real-world applications [58, 73, 92]. In static analysis, an accurate detection may depend on complex program analysis methods [23], e.g., inter-procedural analysis and context-sensitive analysis. These methods establish the global state representation to capture all information of the whole program, which infers the relationships between code in a

relatively accurate way. However, most FaaS applications are written in dynamic languages such as Python and JavaScript [50]. Due to the dynamic features of these languages, building accurate call relationships is an open problem [71, 86], i.e., it is not possible to obtain 100% accurate detection. Therefore, we adopt the optional code backup in this article.

Additional changes or efforts in other language prototypes. For FaaS applications written in different languages, *FaaSLight* may consider some additional customization and optimization.

First, some languages like JavaScript may have no magic functions like Python. Based on this point, we use language-specific functions in *Application Entry Recognition* component to cover this situation to provide flexible customization capabilities for different language features. Second, for the optimization in function-level rewriting, this part needs to implement the on-demand fetching and execution mechanism for optional code. *FaaSLight* may require consideration of the variable scope of code execution under this mechanism for different languages, to guarantee the correctness and availability of FaaS applications.

Overhead analysis of on-demand loading code. We design an on-demand loading mechanism for FaaS applications to fetch and execute the separated optional functions. When executing the optimized FaaS applications, some optional functions may be required during the function execution. In this situation, on-demand loading overhead will increase the function execution time, decreasing the performance optimization space. However, this runtime cost does not affect the warm-start latency of serverless functions, because it is a one-time cost if the container or VM of the serverless function is not released. More details can be found in Section 5.5. Moreover, this overhead is mainly in reading the lightweight file, about 100 ms, which is much smaller than the cold-start latency. In our study, *FaaSLight* has made the application loading latency reduce by 28.78% on average. Thus, it is worth sacrificing a little execution performance overhead for greater overall performance optimization.

Discussion about serverless platforms. Indeed, in ideal conditions where the platform can cache everything effectively, the application code loading latency will not be an issue, and the proposed technique will not be needed. However, in practice, serverless platforms cannot cache everything. Specifically, each function instance or VM has a memory restriction. Moreover, function instances that execute serverless functions are constantly changing. Once new serverless functions are continuously deployed to the serverless platform, cached data in all function instances or VMs needs to be updated, which will cause high maintenance overhead.

Serverless optimization plugin. Serverless Framework's official website has presented the "Serverless Optimize Plugin" [48], which can reduce the file size of AWS Lambda functions written in JavaScript. The main principle of this plugin is to separately bundle required dependency libraries for every serverless function [40], i.e., excluding dependency libraries required by other serverless functions in the same application. Compared with our function-level, fine-grained code analysis approach, this plugin's analysis granularity is at the dependency library level, which is coarser-grained. Moreover, it is challenging to identify optional code relative to the FaaS application, missing opportunities for potential performance optimization.

Potential use of *FaaSLight*. In this article, we propose *FaaSLight* to optimize the cold-start latency of FaaS applications. Besides, *FaaSLight* can also be potentially useful for other software development tasks, such as reducing deployment time. Existing work reported that developers often face the long deployment time of serverless applications and proposed an annotation configuration-based framework named Nimbus [62] to reduce the deployment package size of Java applications on AWS Lambda, thereby reducing the deployment time. Similarly, *FaaSLight* can load only indispensable functions and separate optional functions to reduce the code size and the deployment time.

9 CONCLUSION

We proposed *FaaSLight*, an application-level code analysis approach, to load only indispensable code to optimize the cold-start performance of FaaS applications. Specifically, *FaaSLight* identified the code related to application functionalities by constructing the function-level call graph, and separated other code (called optional code) from the application. The separated optional code can be loaded in an on-demand way to avoid the inaccurate identification of indispensable code causing application failure. *FaaSLight* was implemented as the Python and JavaScript prototypes and evaluated with real-world FaaS applications. Results demonstrated that *FaaSLight* efficiently reduced the application code loading latency (up to 78.95%, on average 28.78%), thereby reducing the cold-start latency. As a result, the total response latency of serverless functions was decreased by up to 42.05% (on average 19.21%). Compared with the state-of-the-art, *FaaSLight* achieved a 21.25× improvement in reducing the average total response latency.

REFERENCES

[1] 2016. Image resize. Retrieved on November 10, 2021 from https://github.com/gxx/aws-lambda-python/tree/master/image_resize.

[2] 2017. Lambda Pandas. Retrieved on November 10, 2021 from https://github.com/nicor88/aws-python-lambdas/tree/master/src/hello_pandas.

[3] 2017. Python OpenCV module for AWS Lambda. Retrieved on November 10, 2021 from https://github.com/aeddi/aws-lambda-python-opencv.

[4] 2017. TensorFlow to AWS Lambda. Retrieved on November 10, 2021 from https://github.com/jacopotagliabue/tensorflow_to_lambda_serverless.

[5] 2018. 2018 serverless community survey: huge growth in serverless usage. Retrieved on May 01, 2022 from https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage.

[6] 2019. AWS Lambda to predict the quality of your wine. Retrieved on November 10, 2021 from https://github.com/pierreant/wine-ml-on-aws-lambda.

[7] 2019. Lxml requests. Retrieved on November 10, 2021 from https://github.com/ryfeus/lambda-packs/tree/master/Lxml_requests.

[8] 2019. OpenCV PIL. Retrieved on November 10, 2021 from https://github.com/ryfeus/lambda-packs/tree/master/Opencv_pil.

[9] 2019. Pandas NumPy. Retrieved on November 10, 2021 from https://github.com/ryfeus/lambda-packs/tree/master/Pandas_numpy.

[10] 2019. Scikit example. Retrieved on November 10, 2021 from https://github.com/mpavlovic/serverless-machine-learning/tree/master/scikit-example.

[11] 2019. Skimage NumPy. Retrieved on November 10, 2021 from https://github.com/ryfeus/lambda-packs/tree/master/Skimage_numpy.

[12] 2019. Terraform AWS Lambda Python. Retrieved on November 10, 2021 from https://github.com/ruzin/terraform_aws_lambda_python/tree/master/example.

[13] 2020. AWS Lambda container image support. Retrieved on May 01, 2022 from https://aws.amazon.com/cn/blogs/aws/new-for-aws-lambda-container-image-support/.

[14] 2020. The CIO's guide to serverless computing. Retrieved on May 01, 2022 from https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing/.

[15] 2020. Sentiment analysis in the Cloud with AWS Lambda. Retrieved on November 10, 2021 from https://github.com/cloudacademy/sentiment-analysis-aws-lambda.

[16] 2020. Serverless Bert hugging face AWS Lambda docker. Retrieved on November 10, 2021 from https://github.com/philschmid/serverless-bert-huggingface-aws-lambda-docker.

[17] 2021. Comparison of cold starts in serverless functions across AWS, Azure, and GCP. Retrieved on May 01, 2022 from https://mikhail.io/serverless/coldstarts/big3/.

[18] 2021. LightGBM Sklearn Scipy NumPy. Retrieved on November 10, 2021 from https://github.com/ryfeus/lambda-packs/tree/master/LightGBM_sklearn_scipy_numpy.

[19] 2021. Python static analysis tools. Retrieved on May 01, 2022 from https://camelcaseguy.medium.com/python-static-analysis-tools-fe5960d8035.

[20] 2021. Top 4 serverless computing platforms in 2021. Retrieved from https://www.loginradius.com/blog/async/serverless-overview/. Retrieved on November 10, 2021.

[21] 2021. Top 5 serverless platforms in 2021. Retrieved on November 10, 2021 from https://www.techmagic.co/blog/top-5-serverless-platforms-in-2020/.

[22] 2021. Vulture—Find dead code. Retrieved on May 01, 2022 from https://github.com/jendrikseipp/vulture.

[23] 2022. Advanced Topics in Programming Languages. Retrieved on September 01, 2022 from https://groups.seas.harvard.edu/courses/cs252/2011sp/.

[24] 2022. Ast for Python. Retrieved on May 01, 2022 from https://docs.python.org/3/library/ast.html.

[25] 2022. Astroid. Retrieved on February 01, 2022 from https://pylint.pycqa.org/projects/astroid/en/latest/.

[26] 2022. AWS Lambda. Retrieved on May 01, 2022 from https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[27] 2022. AWS Lambda function handler. Retrieved on December 03, 2022 from https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html.

[28] 2022. AWS Lambda NumPy. Retrieved on March 17, 2022 from https://github.com/Haodi-Ping/AWSLambda_numpy.

[29] 2022. AWS SAM. Retrieved on November 10, 2021 from https://github.com/aws/aws-sam-cli.

[30] 2022. Azure Functions. Retrieved on May 01, 2022 from https://docs.microsoft.com/en-us/azure/azure-functions/.

[31] 2022. Callgraph. Retrieved on September 01, 2022 from https://github.com/gunar/callgraph.

[32] 2022. Custom-LDA-classifier-model. Retrieved on September 01, 2022 from https://github.com/Haodi-Ping/Custom-LDA-classifier-model.

[33] 2022. Effect size. Retrieved on May 01, 2022 from https://en.wikipedia.org/wiki/Effect_size#Cohen's_d.

[34] 2022. Google Cloud Functions. Retrieved on May 01, 2022 from https://cloud.google.com/functions.

[35] 2022. Google Cloud Functions handler. Retrieved on December 03, 2022 from https://cloud.google.com/functions/docs/writing/write-http-functions.

[36] 2022. Google Cloud Functions numpy pandas. Retrieved on September 01, 2022 from https://github.com/ryfeus/gcf-packs/tree/master/pandas_numpy/sources.

[37] 2022. Google Cloud Functions Python runtime demo. Retrieved on September 01, 2022 from https://github.com/simonprickett/google-cloud-functions-python.

[38] 2022. Google gVisor. Retrieved from https://github.com/google/gvisor. Retrieved on September 01, 2022.

[39] 2022. How can you find unused functions in Python code? Retrieved on May 01, 2022 from https://stackoverflow.com/questions/693070/how-can-you-find-unused-functions-in-python-code.

[40] 2022. How To: Optimize the Serverless Optimizer Plugin. Retrieved on September 01, 2022 from https://www.jeremydaly.com/how-to-optimize-the-serverless-optimizer-plugin/.

[41] 2022. JavaScript async. Retrieved on November 27, 2022. from https://bmaupin.github.io/wiki/programming/javascript/javascript-async.html.

[42] 2022. Mann–Whitney U test. Retrieved on May 01, 2022 from https://en.wikipedia.org/wiki/Mann-Whitney_U_test.

[43] 2022. Microcosm2RSS. Retrieved on September 01, 2022 from https://github.com/conoro/microcosm2rss.

[44] 2022. Python—magic or dunder methods. Retrieved on May 01, 2022 from https://www.tutorialsteacher.com/python/magic-methods-in-python.

[45] 2022. Quotas of Google Cloud Functions. Retrieved on November 27, 2022 from https://cloud.google.com/functions/quotas.

[46] 2022. Serverless Image Processing. Retrieved on September 01, 2022 from https://github.com/unichat-io/serverless-image-processing.

[47] 2022. Serverless-lunch. Retrieved on September 01, 2022 from https://github.com/deekim/serverless-lunch.

[48] 2022. Serverless Optimize Plugin. Retrieved on September 01, 2022 from https://www.serverless.com/plugins/serverless-plugin-optimize.

[49] 2022. Show me a dog. Retrieved on September 01, 2022 from https://github.com/hollandmatt/show-me-a-dog.

[50] 2022. The state of serverless. Retrieved on September 01, 2022 from https://www.datadoghq.com/state-of-serverless/.

[51] 2022. uglify-js. Retrieved on February 01, 2022 from https://lisperator.net/uglifyjs/.

[52] Mikhail.io. 2022. Cold starts in AWS Lambda. Retrieved on May 01, 2022 from https://mikhail.io/serverless/coldstarts/aws/.

[53] Mikhail.io. 2022. Cold starts in Azure Functions. Retrieved on May 01, 2022 from https://mikhail.io/serverless/coldstarts/azure/.

[54] Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. 2018. Practical AJAX race detection for JavaScript web applications. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 38–48.

[55] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards high-performance serverless computing. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*. 923–935.

[56] Gábor Antal, Péter Hegedus, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Static javascript call graphs: A comparative study. In *Proceedings of the IEEE 18th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 177–186.

[57] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*. 263–274.

[58] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. 1697–1714.

[59] Michael D. Brown and Santosh Pande. 2019. Carve: Practical security-focused software debloating using simple feature set mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. 1–7.

[60] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. 135–146.

[61] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ML workflows. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'19)*. 13–24.

[62] Robert Chatley and Thomas Allerton. 2020. Nimbus: Improving the developer experience for serverless applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 85–88.

[63] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*. Springer, 77–101.

[64] 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 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, 467–481.

[65] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. 2022. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Trans. Softw. Eng.* 48, 10 (2022), 4152–4166.

[66] Nafise Eskandani and Guido Salvaneschi. 2021. The wonderless dataset for serverless computing. In *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories*. IEEE, 565–569.

[67] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring serverless computing for neural network training. In *Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. 334–341.

[68] 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 *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. 475–488.

[69] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 363–376.

[70] 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 (ASPLOS'21)*. 386–400.

[71] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2020. Is static analysis able to identify unnecessary source code? *ACM Trans. Softw. Eng. Methodol.* 29, 1 (2020), 1–23.

[72] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless computing: One step forward, two steps back. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research*. Retrieved from www.cidrdb.org.

[73] 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 International Symposium on Software Testing and Analysis (ISSTA'14)*. 248–258.

[74] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. 380–394.

[75] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015. Stateless model checking of event-driven applications. *ACM SIGPLAN Notices* 50, 10 (2015), 57–73.

[76] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[77] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud programming simplified: A Berkeley view on serverless computing. Retrieved from https://arXiv:1902.03383.

[78] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 427–444.

[79] Tofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, Waleed Hashmi, Lakshmi Subramanian, and Yasir Zaki. 2021. Muzeel: A dynamic JavaScript analyzer for dead code elimination in today's web. Retrieved from https://arXiv: 2106.08948.

[80] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *Proceedings of the IEEE 11th International Conference on Cloud Computing*. IEEE, 442–450.

[81] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-efficient microservices on smartNIC-accelerated servers. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*. 363–378.

[82] Xuanzhe Liu, Gang Huang, Qi Zhao, Hong Mei, and M. Brian Blake. 2014. iMashup: A mashup-based framework for service composition. *Sci. China Inf. Sci.* 57, 1 (2014), 1–20.

[83] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. 2007. Towards service composition based on mashup. In *Proceedings of the IEEE International Conference on Services Computing Workshops (SCW'07)*. IEEE Computer Society, 332–339.

[84] Sunil Kumar Mohanty, Gopika Premsankar, and Mario Di Francesco. 2018. An evaluation of open source serverless computing frameworks. In *Proceedings of the IEEE 10th International Conference on Cloud Computing Technology and Science (CloudCom'18)*. 115–120.

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

[86] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. 2018. An extensible approach for taming the challenges of JavaScript dead code elimination. In *Proceedings of IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. 291–401.

[87] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. 2020. Allocation priority policies for serverless function-execution scheduling optimisation. In *Proceedings of the International Conference on Service-Oriented Computing*. Springer, 416–430.

[88] István Pelle, János Czentye, János Dóka, András Kern, Balázs P. Gerő, and Balázs Sonkoly. 2020. Operating latency sensitive applications on public serverless edge cloud platforms. *IEEE Internet Things J.* 8, 10 (2020), 7954–7972.

[89] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. 771–783.

[90] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 193–206.

[91] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. 1733–1750.

[92] 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 Workshop on Forming an Ecosystem Around Software Transformation, FEAST 2017*. 65–70.

[93] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*. 869–886.

[94] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. 2018. A multi-study investigation into dead code. *IEEE Trans. Softw. Eng.* 46, 1 (2018), 71–99.

[95] Josep Sampé, Pedro Garcia-Lopez, Marc Sánchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2020. Toward multicloud access transparency in serverless computing. *IEEE Softw.* 38, 1 (2020), 68–74.

[96] 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 USENIX Annual Technical Conference, (ATC'20)*. 205–218.

[97] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'20)*. 281–295.

[98] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. 329–339.

[99] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the USENIX Annual Technical Conference*. 419–433.

[100] 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*. 1–13.

[101] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'21)*. 138–152.

[102] Davide Taibi, Josef Spillner, and Konrad Wawruch. 2020. Serverless computing-where are we now, and where are we heading? *IEEE Softw.* 38, 1 (2020), 25–31.

[103] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'20)*. 311–327.

[104] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. 2020. Cold start in serverless computing: Current trends and mitigation strategies. In *Proceedings of the International Conference on Omni-layer Intelligent Systems*. IEEE, 1–7.

[105] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaS-Net: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'21)*. 443–457.

[106] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the 14th EuroSys Conference (EuroSys'19)*. 1–16.

[107] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*. 133–146.

[108] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: A systematic review. *ACM Trans. Softw. Eng. Methodol.* (2023).

[109] 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 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 416–428.

[110] Jinfeng Wen, Yi Liu, Zhenpeng Chen, Junkai Chen, and Yun Ma. 2021. Characterizing commodity serverless computing platforms. *J. Softw.: Evol. Process* (2021), e2394.

[111] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'20)*. 30–44.

[112] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE task scheduling for serverless analytics. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*. 653–669.

[113] Michael Zhang, Chandra Krintz, and Rich Wolski. 2021. Edge-adaptable serverless acceleration for machine learning Internet of Things applications. *Softw.: Pract. Exper.* 51, 9 (2021), 1852–1867.

[114] Lei Zhao and Kai-Yuan Cai. 2010. Event handler-based coverage for GUI testing. In *Proceedings of the 10th International Conference on Quality Software*. 326–331.