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

XUANZHE LIU, Peking University, China
JINFENG WEN, Peking University, China
ZHENPENG CHEN, University College London, United Kingdom
DING LI, Peking University, China
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 prominent implementation of serverless computing, representing applications as event-driven and stateless functions. However, existing studies report that functions of FaaS applications suffer severely from cold-start latency. In this paper, 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 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 $21.25 \times$ improvement in reducing the average total response latency.

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 [35, 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, 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 paper, 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 optimizing 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 paper, 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
**FaaSLight**: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing

*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 $21.25 \times$ 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 paper 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., “server-less” 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), 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

---

\(^1\)https://github.com/WenJinfeng/FaaSLight

Manuscript submitted to ACM
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.

Fig. 1. The latency breakdown after requests come in cold and warm starts.

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), 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 paper focuses on the cold-start latency problem. For better illustration, we compare the cold-start latency with the warm-start latency in Fig. 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.

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].

2Fig. 1 shows key latencies, not showing fine-grained latencies like request reception and return due to the black-box feature of commodity platforms.

Manuscript submitted to ACM
**FaaSLight**: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing

## 3 MEASUREMENT STUDY

In this paper, 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**.

### 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 Fig. 1), as well as total response latency. The calculation of these latencies is as follows:

<table>
<thead>
<tr>
<th>App ID</th>
<th>Name</th>
<th>Size (MB)</th>
<th>FC (k)</th>
<th>LoC (k)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>App1</td>
<td>image-resize [1]</td>
<td>64.40</td>
<td>9.66</td>
<td>21.20</td>
<td>resize image and save to Boto3</td>
</tr>
<tr>
<td>App2</td>
<td>lambda-pillow [12]</td>
<td>85.91</td>
<td>11.56</td>
<td>30.68</td>
<td>import Boto3 and Pillow to test</td>
</tr>
<tr>
<td>App3</td>
<td>lambda-pandas [2]</td>
<td>83.88</td>
<td>41.76</td>
<td>151.40</td>
<td>use Pandas to converse data</td>
</tr>
<tr>
<td>App4</td>
<td>scikit-assign [10]</td>
<td>147.71</td>
<td>46.34</td>
<td>160.60</td>
<td>use Sklearn model to predict price</td>
</tr>
<tr>
<td>App5</td>
<td>lxml-requests [7]</td>
<td>25.26</td>
<td>6.57</td>
<td>47.18</td>
<td>use lxml to parse HTML pages</td>
</tr>
<tr>
<td>App6</td>
<td>pandas-numpy [9]</td>
<td>113.36</td>
<td>56.83</td>
<td>192.05</td>
<td>use Pandas and Numpy to generate data</td>
</tr>
<tr>
<td>App7</td>
<td>skimage-lambda [11]</td>
<td>261.47</td>
<td>72.63</td>
<td>267.94</td>
<td>download image and use Skimage to process</td>
</tr>
<tr>
<td>App8</td>
<td>opencv-pil [8]</td>
<td>152.96</td>
<td>35.20</td>
<td>93.71</td>
<td>use OpenCV and PIL to process image</td>
</tr>
<tr>
<td>App10</td>
<td>lightgbm-sklearn [18]</td>
<td>221.70</td>
<td>56.45</td>
<td>216.92</td>
<td>use LightGBM model to predict</td>
</tr>
<tr>
<td>App11</td>
<td>sentiment-analysis [15]</td>
<td>240.92</td>
<td>73.45</td>
<td>280.46</td>
<td>use Sklearn model to predict sentiment statement</td>
</tr>
<tr>
<td>App13</td>
<td>numpay-lambda [28]</td>
<td>79.22</td>
<td>35.69</td>
<td>99.12</td>
<td>use Numpy to converse matrix data</td>
</tr>
<tr>
<td>App14</td>
<td>lambda-opencv [3]</td>
<td>228.70</td>
<td>31.91</td>
<td>74.88</td>
<td>use OpenCV to get properties</td>
</tr>
<tr>
<td>App15</td>
<td>question-answering [10]</td>
<td>2083.58</td>
<td>57.92</td>
<td>197.13</td>
<td>use Bert model to answer questions</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>Max</td>
<td>Max</td>
<td>Max</td>
<td>Max</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td>2083.58</td>
<td>81.91</td>
<td>317.88</td>
<td>Mean</td>
</tr>
</tbody>
</table>

Table 1. The details of our benchmarks.
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 Fig. 2. 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 Fig. 2. It illustrates that the application code loading latency is a significant overhead in cold starts. Generally, the preparation phase latency contains 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

---

3App9 has two main functionalities, i.e., model training (App9-t) and model prediction (App9-p).
its application code loading latency. In practice, our previous experience demonstrated that a similar mechanism is effective to optimize client-side JavaScript applications \cite{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 paper.

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 \cite{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 sub-optimal 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 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.

Fig. 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
Fig. 3. The workflow overview of FaaSLight.

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, 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.

```python
def load_reduce(self):
    stack = self.stack
    args = stack.pop()
    func = stack[-1]
    if len(args) and type(args[0]) is type:
        n = args[0].__name__  # noqa
        try:
            stack[-1] = func(*args)
        return
    except TypeError as err:
        # If we have a deprecated function,
        # try to replace and try again.
        msg = "_reconstruct: First argument must be a sub-type of ndarray"
        if msg in str(err):
            try:
                cls = args[0]
                stack[-1] = object.__new__(cls)
            return
        except TypeError:
            pass
    elif args and issubclass(args[0], BaseOffset):
        # TypeError: object.__new__(Day) is not safe, use Day.__new__()  
        cls = args[0]
        stack[-1] = cls.__new__(*args)
        return
    raise
```

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

```python
def load_reduce(self):
    import custom_functemplate
    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, the "rewrite_template"
method can read this file into memory and fetch the required code in the string format. If it exists, 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 Fig. 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 (R) 4 cores and 24GiB 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].

Fig. 4. The change of statistical values (the optimized application as a percentage of the original application).

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

Ans. 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, Figs. 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 Fig. 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

<table>
<thead>
<tr>
<th>App ID</th>
<th>Version</th>
<th>Preparation phase latency (ms)</th>
<th>Loading phase latency (ms)</th>
<th>Runtime memory (MB)</th>
<th>Total response latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>App1</td>
<td>before</td>
<td>1420.43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1279.10 (~ 9.95%)</td>
<td>748.53</td>
<td>93</td>
<td>4011.32</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1279.38 (~ 10.56%)</td>
<td>701.82 (~ 7.77%)</td>
<td>90 (~ 3.23%)</td>
<td>3792.86 (~ 5.45%)</td>
</tr>
<tr>
<td>App2</td>
<td>before</td>
<td>1483.83</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1298.98 (~ 11.45%)</td>
<td>875.88</td>
<td>68</td>
<td>2485.94</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1279.88 (~ 12.55%)</td>
<td>183.77 (~ 78.95%)</td>
<td>43 (~ 36.76%)</td>
<td>2316.69 (~ 6.17%)</td>
</tr>
<tr>
<td>App3</td>
<td>before</td>
<td>1532.66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1343.44 (~ 12.35%)</td>
<td>1782.62</td>
<td>115</td>
<td>3522.86</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1323.81 (~ 19.56%)</td>
<td>1708.84 (~ 4.77%)</td>
<td>113 (~ 1.74%)</td>
<td>3323.86 (~ 5.65%)</td>
</tr>
<tr>
<td>App4</td>
<td>before</td>
<td>2011.29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1778.86 (~ 11.56%)</td>
<td>2235.33 (~ 4.96%)</td>
<td>142</td>
<td>4593.43</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1766.07 (~ 12.09%)</td>
<td>2033.82 (~ 13.53%)</td>
<td>140 (~ 1.41%)</td>
<td>4163.48 (~ 9.36%)</td>
</tr>
<tr>
<td>App5</td>
<td>before</td>
<td>1279.72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1249.72</td>
<td>451.97 (~ 7.54%)</td>
<td>61</td>
<td>2601.61 (~ 3.27%)</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1272.52</td>
<td>435.84 (~ 10.84%)</td>
<td>60 (~ 3.23%)</td>
<td>2511.24 (~ 6.63%)</td>
</tr>
<tr>
<td>App6</td>
<td>before</td>
<td>1540.93</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1398.48 (~ 9.19%)</td>
<td>1774.72 (~ 9.45%)</td>
<td>125 (~ 8.00%)</td>
<td>3660.67</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1340.51 (~ 12.96%)</td>
<td>1536.38 (~ 21.61%)</td>
<td>107 (~ 14.40%)</td>
<td>3054.51 (~ 16.56%)</td>
</tr>
<tr>
<td>App7</td>
<td>before</td>
<td>2312.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>2185.66 (~ 5.47%)</td>
<td>4217.68 (~ 7.92%)</td>
<td>228 (~ 9.65%)</td>
<td>7165.54</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>2177.49 (~ 5.82%)</td>
<td>1408.62 (~ 69.27%)</td>
<td>130 (~ 42.98%)</td>
<td>6770.75 (~ 5.51%)</td>
</tr>
<tr>
<td>App8</td>
<td>before</td>
<td>1739.22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1642.62 (~ 5.53%)</td>
<td>790.91 (~ 10.92%)</td>
<td>98 (~ 3.92%)</td>
<td>2562.48 (~ 6.51%)</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1620.82 (~ 6.81%)</td>
<td>188.26 (~ 78.80%)</td>
<td>42 (~ 58.82%)</td>
<td>1951.16 (~ 28.82%)</td>
</tr>
<tr>
<td>App9</td>
<td>before</td>
<td>2140.74 (~ 21.90%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>2108.48 (~ 23.08%)</td>
<td>3790.63 (~ 4.89%)</td>
<td>229 (~ 9.04%)</td>
<td>8218.25 (~ 9.04%)</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>3135.82 (~ 21.32%)</td>
<td>216 (~ 6.09%)</td>
<td>7470.49 (~ 17.32%)</td>
<td></td>
</tr>
<tr>
<td>App10</td>
<td>before</td>
<td>2765.90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>2081.80 (~ 12.01%)</td>
<td>2272.51 (~ 4.99%)</td>
<td>229 (~ 6.92%)</td>
<td>8291.30</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1914.79 (~ 19.07%)</td>
<td>1895.94 (~ 20.73%)</td>
<td>215 (~ 6.09%)</td>
<td>7083.05 (~ 14.72%)</td>
</tr>
<tr>
<td>App11</td>
<td>before</td>
<td>2018.32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>1943.81 (~ 3.69%)</td>
<td>3308.98 (~ 2.23%)</td>
<td>122 (~ 24.13%)</td>
<td>6851.93 (~ 2.88%)</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>1949.72 (~ 3.40%)</td>
<td>1722.93 (~ 49.10%)</td>
<td>141 (~ 22.53%)</td>
<td>3934.31 (~ 29.12%)</td>
</tr>
<tr>
<td>App12</td>
<td>Docker</td>
<td>before</td>
<td>1266.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>after1</td>
<td>6966.72</td>
<td></td>
<td>410 (~ 1.91%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>6901.35</td>
<td></td>
<td>410 (~ 1.91%)</td>
</tr>
<tr>
<td>App13</td>
<td>Docker</td>
<td>before</td>
<td>1441.96</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>after1</td>
<td>1252.35 (~ 13.13%)</td>
<td>646.34 (~ 7.85%)</td>
<td>74 (~ 5.13%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>1341.88 (~ 13.88%)</td>
<td>571.64 (~ 18.50%)</td>
<td>69 (~ 11.54%)</td>
</tr>
<tr>
<td>App14</td>
<td>before</td>
<td>2592.21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after1</td>
<td>2153.25 (~ 16.93%)</td>
<td>938.65 (~ 1.54%)</td>
<td>116 (~ 3.13%)</td>
<td>3195.03 (~ 13.13%)</td>
</tr>
<tr>
<td></td>
<td>after2</td>
<td>2032.84 (~ 21.38%)</td>
<td>830.00 (~ 12.94%)</td>
<td>114 (~ 1.72%)</td>
<td>2980.90 (~ 18.96%)</td>
</tr>
<tr>
<td>App15</td>
<td>Docker</td>
<td>before</td>
<td>1306.85</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>after1</td>
<td>1319.33</td>
<td>5923.85 (~ 4.63%)</td>
<td>872 (~ 2.74%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>1307.50</td>
<td>4902.64 (~ 21.07%)</td>
<td>732 (~ 16.06%)</td>
</tr>
</tbody>
</table>

### Average

| Max    | 26.14% | 78.95% | 58.82% | 42.05% |
| Mean   | 11.72% | 28.78% | 14.79% | 19.21% |
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 Fig. 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 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, 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 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 Fig. 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 Fig. 6 and Fig. 7. Program Analyzer part identifies optional functions, which are major consumers of the application code loading latency. Therefore, as shown in Fig. 6 and Fig. 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 Fig. 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.

**Ans. 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 Fig. 8. It guides developers to configure lower billing memory.

**Ans. 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.

Manuscript submitted to ACM
FaaSLight: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing

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.

Ans. to RQ4: The on-demand loading strategy of FaaSLight introduces a small runtime cost (about 100 ms).

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, in order 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 Fig. 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\times 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\times. 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\times 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.

**Ans. to RQ5:** Compared with the state-of-the-art, FaasLight achieves a 21.25\times 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. The 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\times 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.
Table 4. The performance result of FaaS applications written in JavaScript and executed in AWS Lambda (cold starts).

<table>
<thead>
<tr>
<th>App ID</th>
<th>Name</th>
<th>Version</th>
<th>Size (MB)</th>
<th>LoC (k)</th>
<th>Preparation phase latency (ms)</th>
<th>Loading phase latency (ms)</th>
<th>Total response latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>App16</td>
<td>lambda-request [43]</td>
<td>before</td>
<td>4.29</td>
<td>51.32</td>
<td>1377.12</td>
<td>1290.61 (− 6.28%)</td>
<td>355.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>3.38</td>
<td>30.80</td>
<td></td>
<td>303.69 (− 14.56%)</td>
<td>2137.97 (− 9.73%)</td>
</tr>
<tr>
<td>App17</td>
<td>handlebars-fetch [49]</td>
<td>before</td>
<td>10.21</td>
<td>125.91</td>
<td>1119.14</td>
<td>1058.72 (− 11.41%)</td>
<td>229.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>6.67</td>
<td>95.19</td>
<td></td>
<td>187.15 (− 18.38%)</td>
<td>1892.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>187.15 (− 18.38%)</td>
<td>1694.88 (− 10.44%)</td>
</tr>
<tr>
<td>App18</td>
<td>request-cheerio [47]</td>
<td>before</td>
<td>13.03</td>
<td>161.42</td>
<td>1377.60</td>
<td>1084.40 (− 7.91%)</td>
<td>404.42 (− 11.59%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>11.16</td>
<td>111.21</td>
<td></td>
<td>187.15 (− 18.38%)</td>
<td>1833.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>187.15 (− 18.38%)</td>
<td>1699.21 (− 8.34%)</td>
</tr>
<tr>
<td>App19</td>
<td>serverless-image [46]</td>
<td>before</td>
<td>101.95</td>
<td>499.43</td>
<td>970.98</td>
<td>929.47 (− 4.27%)</td>
<td>744.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>97.21</td>
<td>383.52</td>
<td></td>
<td>677.19 (− 9.04%)</td>
<td>2088.38</td>
</tr>
</tbody>
</table>

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, 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 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 Cloud Functions and AWS Lambda, further demonstrating the platform generalization of FaaSlight. Specifically, on Google Cloud 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.

Ans. 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 Fig. 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 [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.
**FaaSLight**: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing

### Table 5. The cold start result of App1 to App19 (originally developed for AWS Lambda) on Google Cloud Functions. App12 and App15 are excluded because they exceed the 500 MB uncompressed size limit of Google Cloud Functions.

<table>
<thead>
<tr>
<th>App ID</th>
<th>Version</th>
<th>Preparation phase latency (ms)</th>
<th>Loading phase latency (ms)</th>
<th>Total response latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>App1</td>
<td>before</td>
<td>2634.28 (~ 12.56%)</td>
<td>1258.78 (~ 10.73%)</td>
<td>6006.80 (~ 12.56%)</td>
</tr>
<tr>
<td>App2</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App3</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App4</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App5</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App6</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App7</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App8</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App9</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App10</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App11</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App12</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App13</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App14</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App15</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App16</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App17</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App18</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
<tr>
<td>App19</td>
<td>before</td>
<td>3090.93 (~ 17.80%)</td>
<td>1410.13 (~ 10.73%)</td>
<td>4401.06 (~ 17.80%)</td>
</tr>
</tbody>
</table>

### Table 6. The details of the newly added FaaS applications executed on Google Cloud Functions.

<table>
<thead>
<tr>
<th>App ID</th>
<th>Name</th>
<th>Version</th>
<th>Size (MB)</th>
<th>Loc (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>App20</td>
<td>google-requests</td>
<td>before</td>
<td>9.09</td>
<td>41.75</td>
</tr>
<tr>
<td>App21</td>
<td>google-pandas</td>
<td>before</td>
<td>12.99</td>
<td>42.93</td>
</tr>
<tr>
<td>App22</td>
<td>LDA-classifier</td>
<td>before</td>
<td>33.53</td>
<td>427.27</td>
</tr>
</tbody>
</table>

**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
Table 7. The cold start result of App20 to App22 (originally developed for Google Cloud Functions) on both AWS Lambda and Google Cloud Functions.

<table>
<thead>
<tr>
<th>App ID</th>
<th>Executed platform</th>
<th>Version</th>
<th>Preparation phase latency</th>
<th>Loading phase latency</th>
<th>Total response latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>App20</td>
<td>AWS Lambda</td>
<td>before</td>
<td>908.06</td>
<td>708.70</td>
<td>2082.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>848.48 (−5.73%)</td>
<td>608.90 (−14.08%)</td>
<td>1890.49 (−9.22%)</td>
</tr>
<tr>
<td></td>
<td>Google Cloud Functions</td>
<td>before</td>
<td>2146.06</td>
<td>345.18</td>
<td>3306.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>2025.58 (−5.61%)</td>
<td>305.85 (−11.39%)</td>
<td>2972.61 (−10.14%)</td>
</tr>
<tr>
<td>App21</td>
<td>AWS Lambda</td>
<td>before</td>
<td>1149.46</td>
<td>3585.32</td>
<td>5137.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>1097.52 (−12.35%)</td>
<td>3072.49 (−17.09%)</td>
<td>4379.61 (−14.77%)</td>
</tr>
<tr>
<td></td>
<td>Google Cloud Functions</td>
<td>before</td>
<td>2476.64</td>
<td>4174.23</td>
<td>7051.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>2112.53 (−14.70%)</td>
<td>3306.28 (−21.17%)</td>
<td>5731.86 (−18.26%)</td>
</tr>
<tr>
<td>App22</td>
<td>AWS Lambda</td>
<td>before</td>
<td>1064.06</td>
<td>1677.95</td>
<td>2834.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>981.71 (−7.74%)</td>
<td>1459.05 (−13.05%)</td>
<td>2547.58 (−10.12%)</td>
</tr>
<tr>
<td></td>
<td>Google Cloud Functions</td>
<td>before</td>
<td>2243.30</td>
<td>2912.45</td>
<td>5972.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after2</td>
<td>2097.86 (−6.48%)</td>
<td>2562.56 (−12.01%)</td>
<td>5408.63 (−9.44%)</td>
</tr>
</tbody>
</table>

* Performance results are in milliseconds.

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], etc. 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
**FaaSLight**: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing

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 paper, 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 paper.

**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, in order 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
Liu et al.

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 paper, 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.

**ACKNOWLEDGMENTS**

This work was supported by the National Key Research and Development Program of China (No. 2020YFB2104100), the National Natural Science Foundation of China under the grant numbers 62172008 and 62172009, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Zhenpeng Chen is supported by the ERC Advanced Grant under the grant number 741278 (EPIC: Evolutionary Program Improvement Collaborators).

We thank the valuable review comments from anonymous reviewers and the timely and kindly review process organized by the associate editor, Prof. Sam Malek. If any concerns are raised, please contact zp.chen@ucl.ac.uk and ding_li@pku.edu.cn for correspondence.

**REFERENCES**


Manuscript submitted to ACM
**FaaSLight**: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing


Manuscript submitted to ACM
Liu et al.


