**LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing**

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, China
HAOYU WANG, Huazhong University of Science and Technology, China
XIN JIN, Peking University, China
XUANZHE LIU, Peking University, China

Serverless computing is an emerging cloud computing paradigm that frees developers from server management. However, existing studies report that software applications developed in serverless fashion (named *serverless applications*) severely suffer from cold start latency. We propose an application-level performance optimization approach called *LambdaLite*, for accelerating the cold start for serverless applications. We first conduct a measurement study to investigate the possible root cause of the cold start problem and find that application code loading latency is the dominant overhead. Therefore, loading only indispensable code from serverless 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 (optional code) from the serverless application. The separated optional code can be loaded on demand to avoid the inaccurate identification of indispensable code causing application failure. In practice, *LambdaLite* can be seamlessly deployed on existing serverless platforms without the need to modify the underlying OSes or hypervisors, nor introduce additional manual efforts to developers. Evaluation results on 15 real-world serverless applications show that our approach can significantly reduce 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 applications can be decreased by up to 42.05% (on average 19.21%). Compared with the state-of-the-art, our approach achieves a $21.25 \times$ improvement on the total response latency of serverless applications.

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

1 INTRODUCTION

Serverless computing is an emerging cloud computing paradigm and has been applied to various domains, including machine learning [49], scientific computing [68], video processing [40], etc. 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 [29], and Google Cloud Functions [31]. With serverless platforms, developers need to only implement their applications (i.e., *serverless applications*) as a set of event-driven functions (i.e., *serverless functions*), each performing an independent task. The underlying serverless platforms automatically handle resource management. Therefore, developers do not need to manage servers or VM instances to run serverless 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 application latency. The resources of idle serverless functions will be released. Therefore, the serverless platform has to initialize the execution environment for most of the invocations to functions that are not frequently used, which prolongs...
the functions’ response latency. In practice, since the latency spent on preparing the execution environment (cold start latency for short) of a serverless function dominates the total response latency [39, 59], it has become one of the performance bottlenecks of modern serverless applications [53, 55, 75, 77]. For example, a previous study [51] reports that the cold start latency can be as much as 80% of the total response latency of serverless applications. Therefore, optimizing cold start latency is a critical challenge of contemporary serverless applications.

Several efforts have been made to optimize the cold start latency at the system level (i.e., optimizing the underlying platforms), such as developing lightweight virtualization technology of containers [45], adjusting the scheduling policy to keep instances warm [72], redesigning sandbox runtime mechanisms [39, 59], etc. 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 to adopt and implement substantial changes to their existing infrastructures. In addition, they also have concerns about security mechanisms, e.g., ASLR [45]. To the best of our knowledge, none of the aforementioned techniques have been applied to commercial serverless platforms.

In this paper, compared to optimizing the cold start latency of serverless applications at the system level, we aim to tackle this problem at the application level. Our guiding principle is to provide a vendor/platform-independent and developer-free technique that application developers can easily adopt to optimize the cold start latency of serverless functions on existing platforms. To achieve our goal, we first investigate the possible root cause of the cold start overhead of serverless applications. We conduct a measurement study on 15 real-world serverless applications. We find that the application code loading latency dominates the cold start latency.

Based on this insight, we propose an application-level approach named LambdaLite, to optimize the cold start latency of serverless applications by reducing the size of executed code, i.e., loading only necessary code. The design principle of LambdaLite 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. LambdaLite does not remove the optional code to guarantee the correctness of the application, 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 serverless applications.

It is worth mentioning that LambdaLite does not introduce any additional manual efforts for developers, nor requires any modification of existing serverless platforms. In practice, LambdaLite can be simply deployed as a service on current serverless platforms. When application developers upload their applications, LambdaLite can process the code optimization and application deployment automatically. In other words, developers are unaware of any changes when LambdaLite works.

We implement LambdaLite as a Python prototype, since Python is one of the most widely used languages in serverless community. We evaluate its effectiveness on 15 real-world serverless applications. The results show that LambdaLite 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 applications can be decreased by up to 42.05% (on average 19.21%). As an additional benefit, LambdaLite can decrease the runtime memory of serverless applications by up to 58.82% (on average 14.79%) due to the reduced size of loaded code. Compared with the state-of-the-art, LambdaLite achieves a 21.25× improvement on the total response latency of serverless applications.

To the best of our knowledge, LambdaLite is the first application-level effort to optimize the cold start latency of serverless applications. 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 applications and find that the application code loading latency is the dominant overhead.
• We propose an application-level performance optimization approach to reduce the cold start latency without compromising the effectiveness, correctness, and availability of serverless applications.
• We evaluate our approach on 15 real-world serverless applications, and the results show that it can significantly reduce the cold start latency.

2 BACKGROUND
In this section, we introduce the background knowledge 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 [73–75], i.e., “server-less” means no server management for developers. Developers focus solely on the business logic of applications. Generally, serverless applications are composed of serverless functions, which are standalone, event-driven, stateless units dedicated to handling specific tasks. 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, 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., containers or other kinds of sandboxes) 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), 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 like AWS S3 [23] 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 of 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. 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 application code (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 [39, 51, 59, 67, 71], and (2) once it happens, its latency dominates the end-to-end response latency of a serverless function [45, 51, 67, 69, 73, 77]. Fuerst et al. [51] 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. [45] 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. [69] 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 [45, 53, 55, 58].

3 A MEASUREMENT STUDY

To further investigate the possible root cause of the cold start latency, we conduct a measurement study on real-world serverless applications executed on AWS Lambda, which is the most popular and widely used serverless platform [20, 21].

1.1 Benchmarks

We select real-world serverless applications from GitHub as our benchmarks according to the following criteria. A serverless application is selected when (1) it is written in Python, which is one of the most widely used languages in

---

1Fig. 1 shows key latencies, not showing fine-grained latencies like request reception and return due to the black-box feature of commodity platforms.
LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing

serverless community [5], (2) its code contains more than 20k lines, indicating a median and large application [65], (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 [28] that is a command-line interface of developing serverless applications. Our final benchmarks consist of 15 real-world serverless applications, ranging from data processing to machine learning. Serverless functions and their dependency libraries are bundled together to implement certain tasks in each serverless application. Specific details are shown in Table 1. In our study, the application size, number of functions, number of lines of code are denoted as Size, FC, and LoC, respectively. FC is calculated by recognizing the number of all function definitions, while LoC is calculated by counting executable statements excluding single-line, multi-line, and document comments. As part of the serverless application, 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 Result

We execute these serverless applications in cold starts, and then obtain their 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:

- **Loading phase latency** is the latency of the application code loading in cold starts. It is extracted from the “Init Duration” attribute provided by AWS Lambda execution logs.
- **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 contained in the serverless application. It is extracted from the “Duration” attribute of AWS Lambda execution logs.
- **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 key phase of the total response latency for serverless applications in Fig. 2. We observe that the application code loading latency is the dominant 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

---

App9 has two main functionalities, i.e., model training (App9-t) and model prediction (App9-p).
execution latency is only 7.57%, on average. Particularly, for 12 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 the dominant 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). Application developers can hardly control the preparation phase latency because commodity serverless platforms such as AWS Lambda are not open for 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 serverless application can optimize the application code loading latency.

Serverless applications are mostly written in high-level languages such as Python [5]. Third-party dependency libraries are often imported to help serverless functions implement complex functionalities. Although developers usually only use a small subset of all supported functionalities of these libraries, all functionalities will still be loaded completely [63, 64]. Thus, eliminating code not used by a serverless application may help optimize its application code loading latency. Since the application code loading latency is a significant part of the cold start latency, which further dominates the end-to-end response latency, optimizing the code size of a serverless application can potentially reduce its overall end-to-end latency. This insight motivates our proposed approach in this paper.

4 APPROACH

Based on our insight, we propose an application-level approach, LambdaLite, that optimizes the cold start latency of serverless applications by eliminating the functions that are not used by the target application (optional functions). Our approach consists of two steps on the high level: (1) build the call graph of a serverless application; (2) eliminate the functions that cannot be reached from the entry points. The key challenge for our approach is to guarantee the correctness of optional function elimination. Serverless applications are often built with dynamic languages like Python [5]. Since building an accurate call graph for dynamic languages is an open problem [52, 60], our method will inevitably misclassify functions that are not optional. Therefore, simply eliminating the functions based on the call graph will inevitably crash the target serverless application in certain cases.

To address this challenge, we first propose a static analysis-based technique that identifies the functions related to the application functionality (indispensable functions). This technique aims to maximize indispensable functions and find optional functions of the serverless application. We do not directly remove optional functions to avoid crashing the target application by eliminating the wrong functions. Instead, we separate optional functions from the application, transform them into string format, and compress them into a lightweight file. Meanwhile, we design a mechanism for serverless applications to fetch and execute optional functions in an on-demand loading way. This mechanism guarantees the availability and correctness of optimized serverless applications on cloud-based serverless platforms.

Fig. 3 shows an overview of LambdaLite, mainly containing three parts (i.e., Preprocessor, Program Analyzer, and Application Generator) with seven components (i.e., ① to ⑦ in the figure). Given a serverless application, first, LambdaLite identifies and removes a part of optional files leveraging Preprocessor part to get a simplified serverless application (Section 4.1). Then, LambdaLite constructs the call graph to generate the final set of indispensable functions for the simplified application through using Program Analyzer part (Section 4.2). Finally, based on the indispensable function set, Application Generator part generates the optimized serverless 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 4.3). Particularly, LambdaLite does not introduce any additional manual efforts for developers, nor modifies any underlying OSes or hypervisors of existing serverless platforms. LambdaLite can be simply deployed as a service on serverless platforms. When application developers upload their serverless applications, LambdaLite automatically optimizes the application code and deploys it to the serverless platform.

![Diagram of LambdaLite](image)

4.1 Preprocessor

Preprocessor part is the prepossessing phase of LambdaLite. It removes files that are not indispensable to get a simplified application.

1. **Optional File Elimination.** According to the actual development process, LambdaLite eliminates four types of optional files for serverless applications. (1) Files related to the local virtual environment. Developers may package some local development files that are not related to the application functionality. For example, “pip” and “setuptools” directories may include in serverless applications; (2) The compiled files like “pyc” or “pyi” files. These files are generated when developers test their applications locally, increasing the package size of serverless applications; (3) The information-related directories in used general libraries. For example, the “dist-info” directory only describes additional information of 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 complex of serverless applications later.

4.2 Program Analyzer

Program Analyzer part is responsible for the core program analysis to obtain the final indispensable functions for simplified applications. It contains five components as follows.

2. **Serverless Function Recognition.** It identifies entry points of the serverless application, i.e., serverless functions. Unlike normal desktop programs, serverless functions are event-driven. Therefore, before eliminating optional functions, LambdaLite first identifies the entry points. The relationship between serverless functions and their events is configured in a global configuration file (e.g., “.yml” in Serverless Framework [35], which is a popular development framework). We can analyze such a file to get serverless functions.

However, sometimes a serverless application may not contain configuration files. For these cases, LambdaLite parses the code and searches for unique representation in the code i.e., parameters in serverless function definitions contain fixed input fields like “event” and “context”. These serverless functions will later participate in the construction process of the call graph.
3. **Magic Function Recognition.** Magic functions are used to define overloaded behaviors in Python programs [34]. These functions are important for maximizing indispensable functions of the serverless application. Because magic functions are executed automatically without being called when specific class operations occur, static program analysis is hard to identify their calling information. The reachable functions of magic functions may be indispensable for serverless applications. Therefore, we need to detect all magic functions to ensure that potentially indispensable functions are found. In Python, magic functions are usually wrapped in double underscore like “__xx__”. LambdaLite parses the code to identify if the function name of all function definitions conforms to the representation of magic functions, and returns satisfying functions to Call Graph Construction component.

4. **Call Graph Construction.** It constructs the call graph of potentially indispensable functions through the call reachability analysis [42, 54, 76]. Our study only pays attention to calling or called relationships of functions. LambdaLite adopts a similar idea of Class Hierarchy Analysis (CHA) [42, 44]. First, it marks entry points to be reachable. Second, definition scopes of all reachable functions are analyzed. For potentially indispensable functions found in a reachable function, they are also marked to new reachable ones and then entered into the analysis iteration process until no more new reachable functions are found. In the final call graph, it contains all reachable functions about entry points and magic functions.

5. **Initial Indispensable Function Generation.** It is to obtain an initial set of indispensable functions related to application functionalities. In the call graph, some magic functions and their reachable functions are unnecessary if the corresponding libraries are not used in serverless applications. Thus, LambdaLite leverages entry points to find their reachable functions in a breadth-first manner from the call graph, and then save them into a set. Based on this set, LambdaLite can determine the used libraries in the serverless application. According to these libraries, LambdaLite extracts the related magic functions and their reachable functions from the call graph, and then appends them into the set, which is regarded as the initial set of indispensable functions.

6. **Special Rule Query.** It is to supplement additional indispensable functions, i.e., pre-loaded functions, and whitelist functions. These functions are useful in the application, but they cannot be identified and analyzed by previous components. Specifically, when a dependency library is imported, its __init__.py file is implicitly executed. Thus, involved functions are automatically loaded and executed, which is hard to capture by static program analysis. In our study, these functions are called pre-loaded functions. LambdaLite establishes a repository for pre-loaded functions related to some common dependency libraries (e.g., Numpy, Scikit-learn) in advance through the dynamic approach offline. The dynamic approach is explained as follows. LambdaLite inserts a print message in the code body of each function definition. This message integrates the location and function name of each function definition. When executing a certain dependency library import, LambdaLite obtains a series of output information and extracts the used functions as pre-loaded functions. Whitelist functions are functions that always be used in certain libraries, but they may be not analyzed by LambdaLite. We set such a whitelist to provide dynamic adjustment.

According to the used libraries obtained from Initial Indispensable Function Generation component, LambdaLite extracts the related pre-loaded functions and whitelist functions, and supplements them to the initial set of indispensable functions to become the final set of indispensable functions.

### 4.3 Application Generator

**Application Generator** part is the application regeneration phase to achieve separation and on-demand loading of optional functions (i.e., functions not in the final set of indispensable functions). We leverage the function-level rewriting operation to achieve this goal.
Function-level Rewriting. It is to separate optional functions from the application and still retain their function definition with empty code body, and rewrite their code body into our custom execution code, which has much fewer lines of code (2 lines) than their original code. Through such a concise way, the loaded code size of the serverless application is reduced. An example of the optional function is shown in Listing 1, whose code body has 23 lines that do not contain comments, and it can be transformed to 2 lines code shown in Listing 2. The custom execution code is to execute the "rewrite_template" method from "custom_funtemplate" module. When rewriting an optional function, if it has the parent function that is also an optional function, the current function will not be rewritten, and later LambdaLite will rewrite its parent function. Such a design can reduce some rewriting code, making the loaded code size smaller.

```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/)

Before rewriting each optional function, its whole function definition is saved in the corresponding value in string format under the key of this function. After handling all optional functions, their key-value content is generated, compressed through a compression strategy (e.g., "gzip"), and saved into a file. When a serverless application is invoked and some optional functions are required, the "rewrite_template" method for these optional functions first checks whether the file is loaded. If it does not exist, read this file into memory and fetch the required code, which is in the form of a string. If it exists, the serverless application can directly fetch the required code and execute it. However, when executing the "rewrite_template" method, it also needs to accept some necessary parameters to help the executions of optional functions. Parameters contain the function name of the required optional function (fetching the corresponding code), callable representation of this optional function (calling this function), external functionalities used in this function (assisting the execution of the code of this optional function), and the number of returned values of this
function (getting and returning output values after execution). These parameters are generated through the code analysis before rewriting the optional function.

```python
1 def load_reduce(self):
2     import custom_funtemplate
3     return custom_funtemplate.rewrite_template("pandas.compatibility.pickle_compatibility.load_reduce", "load_reduce(self)", {"BaseOffset": BaseOffset, "self": self}, 1)
```

Listing 2. An example of the rewritten function.

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

### 4.4 Implementation

**LambdaLite** is implemented as a Python prototype. For each serverless application, **LambdaLite** can generate the corresponding optimized version through seven components. In our approach, the main analysis builds on the foundation and improvements of CHA [42, 44], *ast* [24] and *astroid* [25] libraries.

### 5 EVALUATION

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

- **RQ1 (Code reduction):** How much can **LambdaLite** reduce the size of serverless applications?
- **RQ2 (Cold performance):** How much can **LambdaLite** speed up cold starts of serverless applications?
- **RQ3 (Warm performance):** How does **LambdaLite** affect the performance of warm starts of serverless applications?
- **RQ4 (Overhead analysis):** What is the performance overhead introduced by the on-demand loading mechanism of **LambdaLite**?
- **RQ5 (Comparison):** How does **LambdaLite** perform compared with state-of-the-art methods?

#### 5.1 Evaluation Settings

We evaluate **LambdaLite** with the 15 real-world applications in Table 1. **LambdaLite** 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 serverless applications are executed on AWS Lambda, which is the most popular and widely used serverless platform [20, 21]. In our study, original serverless applications are denoted as *before* applications. Serverless applications processed by *Preprocessor* part of **LambdaLite** are denoted as *after1* applications, and final optimized applications are denoted as *after2* applications.

We run experiments on each of the 15 applications 20 times to collect performance metrics (e.g., latency and memory usage) and use *Mann Whitney U-test* [33, 61] (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 performance of *after2* set is similar to *before* set. We set the threshold of statistical significance as *p*-value < 0.05. We further compute the effect size as the Cohen’s *d* [30], 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 [30].
LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing

5.2 RQ1: Code Reduction

To explore how LambdaLite reduces statistical values, we compare the application package size (Size), 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, LambdaLite makes the size of before applications decrease to their 84.82%, reducing 15.18% Size. For FC and LoC of after2 applications, LambdaLite 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 Preprocessor part can effectively decrease a large part of optional files. Based on after1 versions, LambdaLite leverages Application Generator part to merge and simplified optional functions that are loaded, i.e., rewriting optional parent functions, to further decrease statistical values to get the after2 version. In addition, when executing optimized serverless applications, on average, LambdaLite callbacks only 10 optional functions according to given input cases on demand.

**Ans. to RQ1:** On average, LambdaLite 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 LambdaLite speed 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. Table 2 show their real latency results. 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, LambdaLite reduces 3.40% to 26.14% of the preparation phase latency except for App5, App12, and App15. The application code 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

| App ID   | Version | Preparation phase latency (ms) | Loading phase latency (ms) | Runtime memory (MB) | Total response latency (ms) |
|----------|---------|-------------------------------|---------------------------|--------------------|----------------------------|
| App1     | before  | 1420.43                       | 760.95                    | 93                 | 4011.32                    |
|          | after1  | 1279.10 (- 9.95%)             | 748.53                    | 91                 | 3792.86 (- 5.45%)          |
|          | after2  | 1270.38 (- 10.56%)            | 701.82 (- 7.77%)          | 90 (- 3.23%)       | 3565.94 (- 11.10%)         |
| App2     | before  | 1483.63                       | 873.00                    | 88                 | 2468.94                    |
|          | after1  | 1296.08 (- 11.45%)            | 870.86                    | 68                 | 2316.69 (- 6.17%)          |
|          | after2  | 1279.88 (- 12.55%)            | 837.73 (- 78.95%)         | 61                 | 1602.61 (- 35.09%)         |
| App3     | before  | 1532.66                       | 1794.47                   | 115                | 3522.86                    |
|          | after1  | 1343.44 (- 12.35%)            | 1782.62                   | 115                | 3323.86 (- 5.65%)          |
|          | after2  | 1322.81 (- 19.56%)            | 1708.84 (- 4.77%)         | 113 (- 1.74%)      | 3319.91 (- 11.10%)         |
| App4     | before  | 2011.29                       | 2352.07                   | 142                | 4593.43                    |
|          | after1  | 1778.86 (- 11.56%)            | 2235.33 (- 4.96%)         | 141                | 4163.48 (- 9.36%)          |
|          | after2  | 1768.07 (- 12.09%)            | 2033.82 (- 13.53%)        | 140 (- 1.41%)      | 4084.10 (- 12.83%)         |
| App5     | before  | 1279.98                       | 488.81                    | 62                 | 2589.65                    |
|          | after1  | 1249.72                       | 451.97 (- 7.54%)          | 61                 | 2601.61 (- 3.27%)          |
|          | after2  | 1272.52                       | 435.84 (- 10.84%)         | 60 (- 3.23%)       | 2511.24 (- 6.63%)          |
| App6     | before  | 1540.03                       | 1959.92                   | 125                | 3666.67                    |
|          | after1  | 1398.48 (- 9.19%)             | 1774.72 (- 9.45%)         | 115 (- 8.00%)      | 3346.29 (- 8.59%)          |
|          | after2  | 1340.51 (- 12.96%)            | 1536.38 (- 21.61%)        | 107 (- 14.0%)      | 3054.51 (- 16.56%)         |
| App7     | before  | 2312.04                       | 4580.31                   | 228                | 7165.94                    |
|          | after1  | 2185.66 (- 5.47%)             | 4217.68 (- 7.92%)         | 206 (- 9.65%)      | 6770.75 (- 5.51%)          |
|          | after2  | 2177.49 (- 5.82%)             | 1408.62 (- 69.27%)        | 130 (- 42.98%)     | 4152.73 (- 42.05%)         |
| App8     | before  | 1739.22                       | 887.81                    | 102                | 2741.02                    |
|          | after1  | 1642.62 (- 5.55%)             | 796.91 (- 10.92%)         | 98 (- 3.92%)       | 2562.48 (- 6.51%)          |
|          | after2  | 1620.82 (- 6.81%)             | 188.26 (- 78.80%)         | 42 (- 58.82%)      | 1951.10 (- 28.82%)         |
| App9     | before  | 2741.06                       | 3985.42                   | 230                | 9035.39                    |
|          | after1  | 2140.74 (- 21.90%)            | 3790.63 (- 4.89%)         | 229                | 8218.25 (- 9.04%)          |
|          | after2  | 2108.48 (- 23.08%)            | 3135.82 (- 21.32%)        | 216 (- 6.09%)      | 7470.49 (- 17.32%)         |
| App10    | before  | 2700.32                       | 3828.55                   | 230                | 8291.80                    |
|          | after1  | 2188.82 (- 18.94%)            | 3698.76 (- 3.63%)         | 229                | 7712.55 (- 6.99%)          |
|          | after2  | 1994.47 (- 26.14%)            | 3141.81 (- 17.94%)        | 215 (- 6.09%)      | 7071.03 (- 14.72%)         |
| App11    | before  | 2365.90                       | 2394.77                   | 129                | 4961.16                    |
|          | after1  | 2081.80 (- 12.01%)            | 2272.51 (- 4.99%)         | 158                | 4494.80 (- 9.40%)          |
|          | after2  | 1914.79 (- 19.07%)            | 1895.94 (- 20.73%)        | 148 (- 6.92%)      | 4035.48 (- 18.66%)         |
| App12    | before  | 2018.32                       | 3384.63                   | 182                | 3551.03                    |
|          | after1  | 1943.81 (- 3.69%)             | 3308.98 (- 2.23%)         | 181                | 5407.95 (- 2.58%)          |
|          | after2  | 1949.72 (- 3.40%)             | 1722.93 (- 49.10%)        | 141 (- 22.53%)     | 3934.31 (- 29.12%)         |
| App13    | before  | 2266.90                       | 6964.72                   | 410                | 8442.14                    |
|          | after1  | 2055.25                       | 6091.35                   | 410                | 8281.06 (- 1.91%)          |
|          | after2  | 2179.19                       | 6036.30 (- 13.36%)        | 397 (- 3.17%)      | 7448.55 (- 11.77%)         |
| App14    | before  | 2141.96                       | 704.41                    | 78                  | 2304.29                    |
|          | after1  | 2153.25 (- 16.93%)            | 938.65 (- 1.54%)          | 116                | 3195.03 (- 13.13%)         |
|          | after2  | 2032.94 (- 21.58%)            | 830.00 (- 12.94%)         | 114 (- 1.72%)      | 2980.90 (- 18.96%)         |
| App15    | before  | 1368.85                       | 6211.63                   | 372                | 8906.10                    |
|          | after1  | 1319.33                       | 5923.85 (- 4.63%)         | 872                | 7840.81 (- 2.74%)          |
|          | after2  | 1307.50                       | 4902.64 (- 21.07%)        | 732 (- 16.06%)     | 6635.88 (- 17.69%)         |

Table 2. The performance result of serverless applications in cold starts.
LambdaLite reduces the preparation phase latency by up to 26.14% (11.72% on average).

For the loading phase latency, Application Generator part is an effective way to decrease this latency by rewriting optionally loaded functions as ones with only two lines of code. Specifically, LambdaLite makes the loading phase latency reduced by up to 78.95%. On average, serverless applications have 28.78% performance improvement on the loading phase latency. Especially for App2, App7, and App8, LambdaLite 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, LambdaLite makes the total response latency reduced by up to 42.05% (19.21% on average). 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 applications have a statistically different performance from their original applications. In these applications, eight applications (App1, App2, App4, App6, App7, App8, App9-t, App9-p) show large effect sizes (>= 0.8), i.e., large performance improvement effect. Five 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 applications are nearly 0.8, indicating that they have a relatively large performance improvement effect at the medium effect level. Only one 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, LambdaLite can significantly improve the total performance of serverless applications.

Table 3. Statistical test results of the total response latency between after2 and before applications in cold starts. The symbol “*” represents that the p-value is less than 0.05.
As shown in Fig. 4, *after1* applications that apply *Preprocessor* part of *LambdaLite* can reduce more optional files. However, these optional files are mostly from files unrelated to the application loading code. Directly deleting such files helps serverless 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. *Application Analyzer* part identifies all 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.

*LambdaLite* 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 applications (e.g., App12 and App15) have the “big” deployment package size, exceeding the normal deployment size limit. They are deployed by the container image. We find that the billed duration of this 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 *LambdaLite* reduces 13.34% to 20.71% billed duration of heavy serverless applications like App12 and App15.

**Ans. to RQ2:** *LambdaLite* 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...
Moreover, the performance improvement achieved by LambdaLite is statistically significant for 93.33% of the studied serverless applications. As an additional benefit, LambdaLite decreases the runtime memory by up to 58.82% (on average 14.79%).

5.4 RQ3: Warm Performance

To explore the effect of LambdaLite on warm starts, we compare the scheduling phase latency, total response latency, and runtime memory of serverless applications. We find that LambdaLite 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 have not statistically different performances than original ones in warm starts. We further explore the runtime memory, and find that LambdaLite 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: LambdaLite has no observable effect on the performance of serverless applications in warm starts and reduces the runtime memory by up to 57.84% (on average 14.74%).

5.5 RQ4: Overhead Analysis

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

We measure the runtime cost introduced by the on-demand loading strategy to answer this research question. We find that its latency overhead 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 on-demand loading does not affect the warm start latency of serverless applications because it is a one-time cost if the container or VM of a serverless function is not released. LambdaLite loads all optional functions when the first optional function is invoked. 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. Therefore, the other nine requests will not be affected. Note that the ten requests consist of one cold start and nine warm starts. Since LambdaLite can reduce the cold start latency by 1,000 ms on average, it is beneficial to trade the one-time 100 ms execution latency for the reduction of the cold start latency.

Ans. to RQ4: The on-demand loading strategy of LambdaLite introduces a small performance overhead (about 100 ms).
To further demonstrate the effectiveness of LambdaLite, 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, 32]. Vulture identifies the objects that have been defined but not used in all given Python files, and reports them as optional code. Vulture and LambdaLite 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 Preprocessor part and Vulture. Optional functions identified by Vulture are separated and rewritten by Application Generator part of LambdaLite, in order to be able to load them on demand to ensure the availability of the optimized serverless application. Results of these methods are shown in Fig. 9.

We first compare the performance improvement of Vulture method and LambdaLite. On average, Vulture method shows 0.90% performance improvement on the total response latency, while LambdaLite can achieve 19.21% improvement. It illustrates that the latency improvement of LambdaLite is 21.25× that of Vulture. In addition, in the best case, Vulture method obtains 3.69% performance improvement, while LambdaLite achieves 42.05% improvement. Moreover, Vulture method does not achieve performance improvement on some serverless applications, such as App9, App11, App13, App14. The reason is that the number of optional functions verified by Vulture is small (only 400 on average). When the optimized serverless 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, LambdaLite removes some optional files through Preprocessor part, and separates on average 5,000 optional functions that are loaded through Program Analyzer part. Thus, LambdaLite shows the effective performance improvement in all tested serverless applications.

We further analyze the principle of Vulture. It identifies only the function objects that have been defined but not used in code. We find that 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 serverless applications. On the contrary, LambdaLite 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 serverless 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 Preprocessor part can be analyzed. The mixed method obtains
a 7.09% improvement of the total response latency on average. It illustrates that our Preprocessor part has a positive impact on performance improvement, speeding up Vulture 7.85×. Compared with the mixed method and LambdaLite, the ability to identify optional functions of loaded code can be analyzed for Vulture and our Program Analyzer part. Results show that LambdaLite improves by 12.12% on the basis of the mixed method. In this situation, the improvement of the total response latency of LambdaLite 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 LambdaLite have a positive impact on optimizing the total response latency of serverless applications.

**Ans. to RQ5:** Compared with the state-of-the-art, LambdaLite achieves a 21.25× improvement on total response latency.

6 THREATS TO VALIDITY

**Internal validity.** In the measurement study, we explore the possible root cause of the cold start overhead of serverless applications. Since serverless 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 mitigate this threat, we conduct 20 measurements for each tested serverless application. Then, we adopt the average value among measurements as the final latency result of the application. Similarly, for the experimental evaluation, we also measure 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 serverless 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, LambdaLite adopts the strategy of identifying as many indispensable functions as possible. Moreover, we also design a mechanism for serverless 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 serverless applications while guaranteeing the correctness and effectiveness of serverless applications.

**External validity.** In LambdaLite, we design the Special Rule Query component to supplement the pre-loaded functions of the used libraries to the final set of indispensable functions. The incompleteness of the repository for pre-loaded functions may result in not being able to add pre-loaded functions for all related libraries. To mitigate this threat, we design an offline dynamic approach to allow LambdaLite to generate pre-loaded functions of any required library. In future work, we will analyze the libraries commonly used by serverless applications, generate corresponding pre-loaded functions for them, and update the repository.

In addition, we evaluate LambdaLite with 15 real-world serverless applications executing on AWS Lambda. This may lead to the limited generalizability of LambdaLite to serverless applications executed on other existing serverless platforms. However, serverless computing allows application developers to benefit from the event-driven feature and focus on only the application logic, without adding too much additional programming effort. Moreover, different serverless platforms follow the same programming abstract. Therefore, LambdaLite is widely applicable to serverless applications executed on different serverless platforms.

7 RELATED WORK

**Serverless computing.** Serverless computing has been used in a wide range of software applications [40, 50, 62, 68], and thus attracted increasing attention from the SE community [38, 41, 43, 46–48, 56, 57, 66, 70, 74, 75]. Some measurement
studies [17, 75] have been presented to help developers select the most appropriate serverless platform, and the multi-cloud approach [66] was designed to make developers fully enjoy the benefits of serverless computing. Lenarduzzi et al. [56, 57] investigated the technique debt affecting serverless applications from the aspect of architecture, code, testing, etc., and discussed the difficulty and possible practices of testing and debugging on serverless computing. To facilitate developers develop their serverless applications, Wen et al. [74] uncovered 36 specific challenges that developers encounter in developing serverless applications. Moreover, a comprehensive study about serverless applications [46] was presented by SE researchers to show specific usage characteristics. A new programming framework called Crucial [41] was presented to execute serverless applications that require fine-grained support for mutable shared state and synchronization. In our study, we present an application-level code analysis approach to optimize the code of serverless applications. This approach can be adopted by developers to improve the cold start latency of serverless applications.

**Cold start optimization.** To reduce the number of cold starts, 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 [36, 37]. 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 serverless applications. Moreover, developers can easily identify this policy, causing them to keep resources warm by making frequent dummy invocations. This practice exacerbates the resource waste problem. Except for the “keep-alive” policy, some studies about cold starts have presented new systems by optimizing the underlying platforms [39, 45, 51, 59, 72]. 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 applications without any additional overhead.

8 CONCLUSION

In our study, we presented the first work that optimized the cold start latency of serverless applications at the application level. We proposed **LambdaLite**, an application-level code analysis approach, to load only indispensable code to optimize the cold start latency of serverless applications. Specifically, **LambdaLite** identified the code related to application functionalities through 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. **LambdaLite** was implemented as a Python prototype and evaluated with 15 real-world serverless applications. Results demonstrated that **LambdaLite** 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 applications was decreased by up to 42.05% (on average 19.21%). Compared with the state-of-the-art, **LambdaLite** achieved a 21.25× improvement on the total response latency of serverless applications.

REFERENCES

[1] 2016. Image resize. https://github.com/gxx/aws-lambda-python/tree/master/image_resize. Retrieved on November 10, 2021.
[2] 2017. Lambda Pandas. https://github.com/nicor88/aws-python-lambdas/tree/master/src/hello_pandas. Retrieved on November 10, 2021.
[3] 2017. Python OpenCV module for AWS Lambda. https://github.com/aeddi/aws-lambda-python-opencv. Retrieved on November 10, 2021.
[4] 2017. TensorFlow to AWS Lambda. https://github.com/jacopotagliabue/tensorflow_to_lambda_serverless. Retrieved on November 10, 2021.
[5] 2018. 2018 serverless community survey: huge growth in serverless usage. https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage. Retrieved on May 01, 2022.
[6] 2019. AWS Lambda to predict the quality of your wine. https://github.com/pierreant/wine-ml-on-aws-lambda. Retrieved on November 10, 2021.
[7] 2019. Lxml requests. https://github.com/ryfeus/lambda-packs/tree/master/lxml_requests. Retrieved on November 10, 2021.
LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing

[71] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba Cloud Function Compute. In Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021. 443–457.

[72] 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 Fourteenth EuroSys Conference 2019, EuroSys 2019. 1–16.

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

[74] 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 20th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021. 416–428.

[75] Jinfeng Wen, Yi Liu, Zhenpeng Chen, Junkai Chen, and Yun Ma. 2021. Characterizing commodity serverless computing platforms. Journal of Software: Evolution and Process (2021), e2394.

[76] Renjun Ye, Liang Liu, Simin Hu, Fangzhou Zhu, Jingxiu Yang, and Feng Wang. 2021. JSLIM: Reducing the known vulnerabilities of JavaScript application by debloating. In Proceedings of the International Symposium on Emerging Information Security and Applications, EISA 2021. Springer, 128–143.

[77] 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 2020 ACM Symposium on Cloud Computing, SoCC 2020. 30–44.