LogLess:

A Logging Paradigm for Serverless Applications

Kushagra Soni
Master of Computer Science
University of Illinois
Urbana Champaign, Illinois, USA
soni14@illinois.edu

Abhishek Shinde
Master of Computer Science
University of Illinois
Urbana Champaign, Illinois, USA
ashinde2@illinois.edu

Simranjit Bhamra
Master of Computer Science
University of Illinois
Urbana Champaign, Illinois, USA
sbhamra2@illinois.edu

Abstract

Logging is a prominent practice followed in software development. Without sufficient and appropriate logging, isolating bugs, debugging problems, and maintaining best security practices become increasingly difficult.

Current logging mechanisms in serverless platforms can sometimes become cumbersome for developers. In case of any failures or debugging, spotting the issues can become difficult due to inadequate logging. Additionally, the auto-generated system-wide logs, by cloud platforms, are not descriptive enough to help debug an application error. There are indeed some logging tools and utilities available in the market, which can be configured to an application's needs. However, they still need developers to write the logging statements apart from the code.

We propose LogLess, a new paradigm in logging, which can automate the logging mechanism. With this approach, the developers will be able to publish logs of their serverless applications without writing the log statements manually within the code. LogLess utilizes the "decorator" objects in a programming language. These decorators will be used to parse the function arguments and every underlying block or line of code within the function. LogLess will categorize every block/line of code based on the function it is performing such as variable assignment, API call request/response, connection to external databases, etc. Upon categorizing these lines of codes, LogLess will assign them to a corresponding "pattern group/sub-group". Once the pattern group has been assigned, it will fetch the corresponding values of all the underlying variables in each block/line of code and generate an appropriate log.

Keywords

Cloud Computing, Distributed Systems, Serverless Computing, Logging, Programming Languages, Decorators

I. INTRODUCTION

LogLess is a one-of-a-kind logging model and makes use of the fact that functions are "first-class objects" in programming languages, like Python, JavaScript, Java, Golang, etc. This allows functions to be passed as arguments to decorators which are functions themselves, to be used by LogLess. LogLess advances knowledge by offering a unique perspective and approach to how logging can be produced. In contrast to the traditional methodology of writing time-consuming logging statements in every block of code, this proof of concept offers a *log-less* decorator-based approach that will automate a range of logging methods for serverless applications. The LogLess being the logging mechanism for Serverless applications, needs to able to provide logging support for the functions which are executed in a request-response fashion. Most of the applications currently being developed on the Serverless architecture, do not support persistent data and hence, any logging novel solution needs to take this limitation in mind. LogLess being approached as an idea specifically for serverless applications can provide a quick, customizable, and developer-friendly logging model.

Currently, developers write individual logging statements in code blocks which could lead to missing key statements or heavily duplicate messages across the platform. LogLess would offer a contribution that simplifies logging by enabling developers to focus on their code. LogLess would enforce standardization on statement messages so that they can be reused.

Different development environments may require differing requirements for logging. LogLess will provide a novel contribution in that a logging mode can be selected from a list of modes. A logging mode will vary in terms of the number of log statements, what parameters get logged, and which log levels are supported. In addition, various modes can be

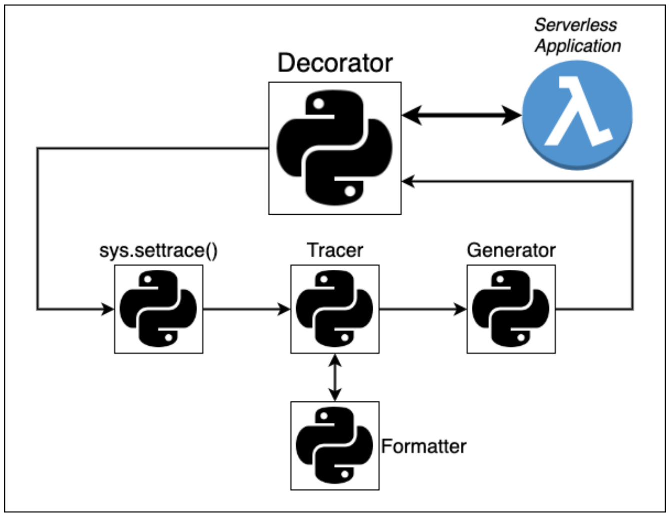


Figure 1: Workflow diagram of LogLess. It shows all the major components which enable the automated Log generation process

configured namely – dev-mode, safe-mode, and prod-mode. Each mode will serve a different purpose in the software development lifecycle and won't need any additional changes to the actual code for debugging and logging. This configuration can provide better logging flexibility and design to applications.

Many serverless platform providers have predefined logging of their cloud services. For example, in AWS, a connection between a lambda function and DynamoDB is logged into CloudWatch. However, this logging is restricted to only AWS-provided services and is not inclusive of any external communications. Moreover, CloudWatch logs do not provide in-depth knowledge of variables and values used within a Lambda application or offer any customization opportunities. LogLess will offer more efficient, granular, and customized logging that will augment such in-built logging tools.

This project can provide a broader benefit to the serverless development community, which includes computer science students, development teams in the workplace, and open-source contributors. While we plan to experiment with LogLess using Python, it can be implemented in other programming languages, thus impacting a larger community.

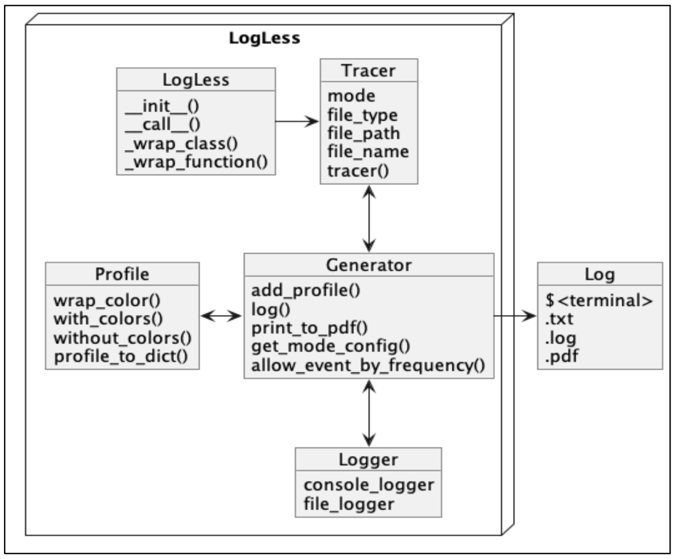


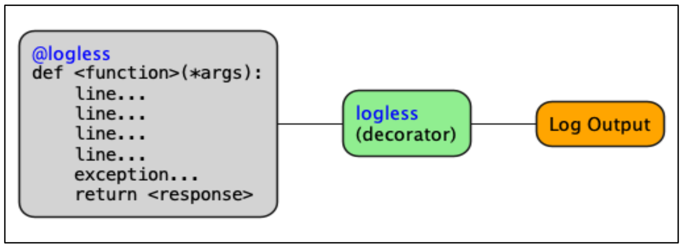
Figure 3: LogLess Object Model of all the components with respective elements and methods

II. THE COMPONENTS

In most modern programming languages, such as Python, Java, and JavaScript, functions are first-class citizens and can be passed around as variables. This means, all these objects can have attributes, member functions, and they can be passed as arguments.

Using this knowledge about functions we implement LogLess by developing a Python *Decorator* function which wraps the application code (decorated function).

Through the *decorator*, we implement a programming concept called "tracing". Python's *sys* module provides some of the most powerful functions among which is *sys.settrace()*. [1]. The usage of sys.settrace() method requires that it should be invoked just before call operation of a function which is to be traced. The settrace() method takes a tracer method as an object. This tracer() method and its implementation has been explained in *$III.A*.



```
@logless
def <function>(*args):
    line...
    line...
    line...
    line...
    exception...
    return <response>
```

Figure 2: Usage of LogLess utility on a function, Its passed as a decorator

A. LogLess

- The *LogLess* is the driver module which comprises of several components. Figure 2 shows the various objects that form *LogLess* library. It integrates all the underlying functionalities. The *LogLess* module defines the functionality of the decorator which in-turn triggers the logging mechanism. This class is responsible for running the *Tracer* module which implements the tracing mechanism. An instance of It is passed as an argument to the *sys.settrace()* function.

B. Tracer

- This module implements code tracing functionality in LogLess. The *Tracer* module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It is passed as an argument to the *sys.settrace()* function. Its implementation returns itself and passes relevant information to the *Generator* module which in turn produces the Log Output. This program goes through each code frame one by one and sends relevant information ($III.B) to the generator.

C. Generator

- This module is executed by the Tracer for each event type, like call, line, return, etc. For each event type, the Generator module produces a relevant logging record which has information like *timestamp*, *log level*, *logger name* and *message* generated. The Generator module utilizes *Profile* and *Logger* modules to format and generate the appropriate logging.

D. Profile

- This module determines the *pattern*, *color-code* and *format* of the output log statements. It utilizes the various properties passed as arguments to the decorator (*$III.A*) to profile the code frames of each event type.

E. Logger

- This is the final logging statement which is produced by LogLess using Tracer, Generator and Formatter modules. The output can be console output or a file, based on user preference.

III. SYSTEM DESIGN

The entry point to the *LogLess*, is a *decorator* [1] which utilizes the fact that functions are "first-class objects". What this means is that a function can be passed around and used as arguments just like any other object example, string, float, etc. As shown in the *Figure 3*, once a function is "decorated" with a "decorator" or passed as an argument, that function is called a *decorated function*.

A. Decorator and the Decorated Function

The *decorator* can be invoked using the @ operator on top of another function or by invoking as a regular function with another function passed as an argument. It utilizes an *inner function* [2] to perform any kind of operation on the *decorated function* passed as argument. The *Inner function* or *nested functions* are defined inside another function. They are used to protect any operation from outside the outer function (or decorator in this case).

Once the decorator is invoked, the *decorated function* and its corresponding arguments are passed to inner function via decorator as shown below:

```
def decorator(function):
   def wrapper(*args, **kwargs):
      <operation on function(*args, **kwargs)>
   return wrapper
```

The inner function or wrapper can perform any required operation on the decorated function, like invoking the function and this the feature that *LogLess* utilizes. It invokes the function inside the wrapper and just before the invocation, starts a tracing routine for the entirety of decorated function's execution workflow. For tracing, LogLess uses the sys.settrace() module comes along with Python's Standard distribution.

Along the with the decorated function, you can pass more arguments to the decorators. These arguments can either be passed enclosed in a parenthesis when decorator is invoked in one of the following ways:

- Using @ operator

```
@decorator (*args)
def function():
      <function body>
```

- Using decorator by passing the function as argument

```
def function():
      <function body>

decorator(logless, *args)
```

LogLess provides multiple execution modes for generating logs namely – SAFE, DEV and PROD. These modes can be passed as argument to the logless decorator as *mode=DEV*. There are other optional features like – *file_type*, *file_path*, *file_name*.

B. Tracing and the Tracer

To create automated logging, we need to capture the internals of the decorated function. This can be achieved by using a powerful tool provided by python language, *sys.settrace()*. The sys.*settrace()* [3] module registers the traceback to the Python interpreter. A traceback is the information that is returned when an event happens in a code, like calling of a function, execution of a line code, return, exception, etc.

```
settrace(tracer)
result = function(*args, **kwargs)
settrace(None)
```

The sys.settrace() function sets the system / global trace function, which allows you to implement a Python source code debugger in Python. It must have three arguments – frame, event, arg, and returns a reference to the local trace function.

- Syntax(s):
  - o sys.settrace(frame,event,arg.frame)
  - o sys.settrace(<local tracer function object>)
- Parameters:
  1) frame: frame is the current stack frame
  2) event: A string which can be either 'call', 'line', 'return', 'exception' or 'opcode'
  3) arg: Depends on the event type
- Returns: Reference to the local trace function which then returns reference to itself.

The trace function is invoked (with event set to 'call') whenever a new local scope is entered; it should return a reference to a local trace function to be used for the new scope, or None if the scope shouldn't be traced. The local trace function should return a reference to itself (or to another function for further tracing in that scope), or None to turn off tracing in that scope. If there is any error occurred in the trace function, it will be unset, just like settrace(None) is called.

The events which trigger the trace have the following meaning:

- call: When a function is invoked, the global trace function is called. The *arg* parameter is *None*; the return value specifies the local trace function.
- line: When a new line of code is about to be executed by the interpreter or a condition of a loop is re-executed. The local trace function is called, with *arg* as *None*; the return value specifies the new local trace function.
- return: When a function (or other code block) is about to return. The local trace function is called; *arg* is the *return value*. It will be *None* if the event is caused by an exception being raised. The trace function's return value is ignored.
- exception: When an exception has occurred. The local trace function is called; *arg* is a *tuple* (*exception*, *value*, *traceback*); the return value specifies the new local trace function.

The tracer also invokes the appropriate Generator logging method based on the Log Mode (refer *$III.E*), supplied by the user to the logless decorator.

C. *Frame, Code and more*

When the local trace function is invoked for each event, the *frame*, *event* and *arg*, arguments are passed to it. We make use of these objects and setup a mechanism to extract the information they contain. The frame contains various information, for example:

- *f_code:* frame code
- *f_code.co_name:* calling function name
- *f_lineno*: line number

Using the frame and event type, we formulate a mechanism of capturing the nuances of the code being executed and capture certain attributes to make a *Profile* of the frame code being traced.

D. *Creating Code Profile*

The Profile module takes the below mentioned attributes from tracer() as input and creates a Profile instance of each event frame being traced.

- Event type
- Variable Name
- Variable Value
- Return Value
- Exception

The profile module is also responsible for formatting and color coding the PDF and console modes of the log statements based on the attributes like event type, log level, etc. This profile object is then passed to the Generator module which utilizes loggers to generate log statements as per the default or user selected options on the decorator.

E. *Generating the Logs*

The *tracer* method creates an instance of *Generator* class for every event being traced. Once generator is created, the tracer uses the logging *mode* information supplied by the decorator module and invokes the corresponding logging method using generator object.

LogLess provides various types of configurable logging options. This gives the serverless application developers an ease to focus on the application rather than creating logging statements. These options are: Mode, File Type, File Path, File Name.

1) Mode:

   LogLess is configured to produce logs in 3 different modes.

   a) SAFE mode:

      - This is also the default mode set in LogLess. When the logless is executed in this mode, the generator will produce logging statements with hidden variable or return values. This way the execution flow of the application code can be tracked in a safe mode without revealing the underlying values of the various variable initializations, assignments, or function returns.
      - Another property of this mode is that it will only log, INFO, DEBUG and ERROR log levels.
      - It can be used by passing "*mode='SAFE'*" as an argument to the *@logless.log* decorator.

   b) DEV mode:

      - This mode works in the similar way as the SAFE mode, but it's configured to reveal the values of all the variables and returns being logged.
      - Along with that it supports more log levels namely, INFO, DEBUG, WARNING, ERROR, CRITICAL.
      - It can be used by supplying *"mode='DEV"* as an argument to @logless.log() decorator.

   c) PROD mode:

      - This mode works in the similar way as the DEV mode, but it's configured to hide the values of all the variables and returns being logged.
      - Along with that it supports more log levels namely, INFO, DEBUG, WARNING, ERROR, CRITICAL.
      - It can be used by supplying *"mode='PROD"* as an argument to @logless.log() decorator.

2) File Type:

   With default logging on the console, LogLess decorator also provides an optional argument which can be used to pass the type of log file user wants to generate.

   The currently supported file extensions which logless generates are:

   - .txt → regular text file which is not color coded.
   - .log → standard log file which is very similar to .txt, but most of the editors and code readers provide good readability support for .log extension files.
   - .pdf → this version of logs will have color coded logging statements.

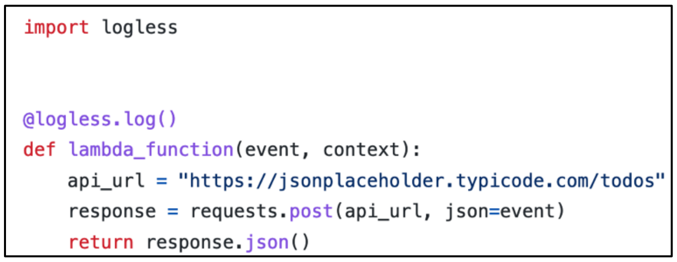
3) File Path & File Name:

   - These options are for providing a user defined names to the log files if the user opts for generating a File Type logging.
   - The File Path is relative to the executing application script to which the logless decorator is being used in.
   - The File Name is the name to be given to the logging file once its generated.

IV. The Usage

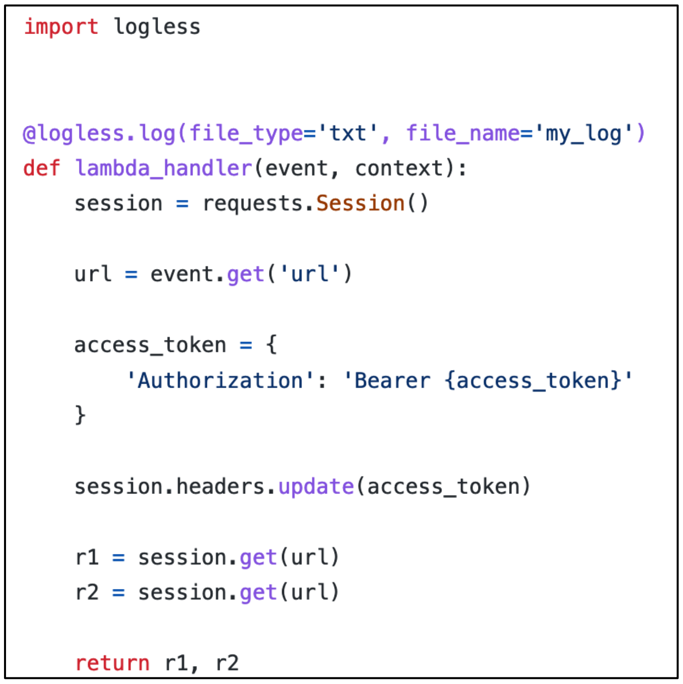
To utilize the LogLess functionality, developers need to "decorate" their application code methods/functions with a unique decorator named - *@logless.log()*. To start with the *LogLess* package needs to be installed and *logless* module is to be imported in the application code.

Once the logless module is imported, the decorator functionality can be used in various ways
as shown below.



```python
import logless


@logless.log()
def lambda_function(event, context):
    api_url = "https://jsonplaceholder.typicode.com/todos"
    response = requests.post(api_url, json=event)
    return response.json()
```

Figure 6: A sample serverless lambda application code which takes an event argument, sends a REST call to an endpoint and send its response back in JSON format



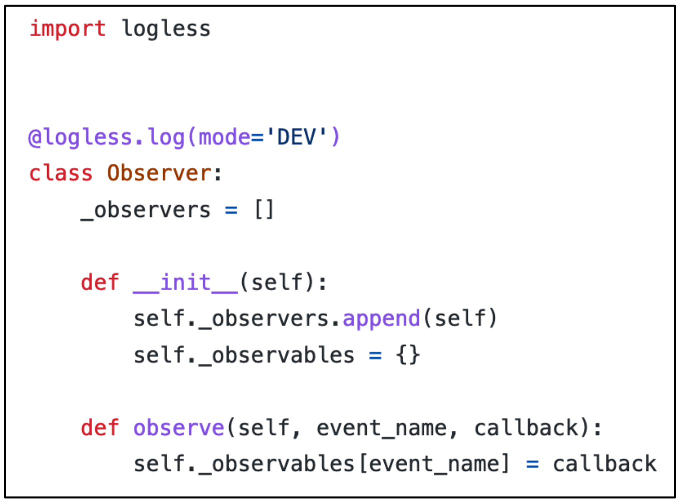
```python
import logless


@logless.log(file_type='txt', file_name='my_log')
def lambda_handler(event, context):
    session = requests.Session()

    url = event.get('url')

    access_token = {
        'Authorization': 'Bearer {access_token}'
    }

    session.headers.update(access_token)

    r1 = session.get(url)
    r2 = session.get(url)

    return r1, r2
```

Figure 5: A sample serverless lambda utilizes the file_type and file_name options of the logless..

V. EXPERIMENTATIONS

A. Benchmarking

To evaluate LogLess performance we set up various benchmarking experiments. These experiments include three main groups, applications with LogLess, plain applications – those with no logs, and applications with manual logs. To prepare these applications we made variations according to each group type. The test applications used for benchmarking are app1.py, app2.py, app3.py, app4.py, and app5.py. The benchmarks are each recorded for 100, 1000, 2000, 3000, 4000, and 5000 runs, and results are captured in seconds. With a benchmarking script, we created the following statistics



```python
import logless


@logless.log(mode='DEV')
class Observer:
    _observers = []

    def __init__(self):
        self._observers.append(self)
        self._observables = {}

    def observe(self, event_name, callback):
        self._observables[event_name] = callback
```

Figure 4: A sample serverless lambda application code uses the mode='DEV' option of the logless

captured: max, mean, median, min, q25, q75, std, and average time taken.

B. Unit Tests & Application Execution

In order to set up our unit testing suite, we leveraged the pytest and pytest-mock packages because pytest is a mature testing framework and pytest-mock enables us to mock out dependencies that our functions utilize. We decided to utilize the coverage library to measure code coverage and grow the testing suite in an effective way.

In addition to unit tests, we decided to manually test and evaluate the execution of Logless on seven different lambda applications. To prepare this manual experimental setup, we used the *python-lambda-local* package for local lambda execution.

VI. ANALYSIS

A. Benchmarking

As LogLess is an additional decorator added to an application we expect the application to add additional runtime. Figure 7 and 8 show the mean runtime in seconds of all five applications with and without LogLess enabled. Looking at these results, we can conclude that the addition of LogLess does not add significant runtime to the applications. Across all applications tested, LoggLess added less than ten milliseconds to the total application runtime, compared to the same application with no logging at all. Similarly, in figure 9, we see that LogLess adds less than four milliseconds to the application runtime when compared to static manual logging.

While the addition of LogLess to an application produces a small increase in runtime, the benefits of automatic, detailed information captured by LogLess make this a fair tradeoff for most use cases. For tasks where response time is a critical priority, there is potential that LogLess could be extended to provide asynchronous logging without impacting round-trip processing time.

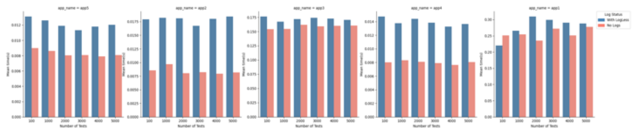


Figure 8: Bar graph of mean time in seconds for apps with LogLess and No Logs

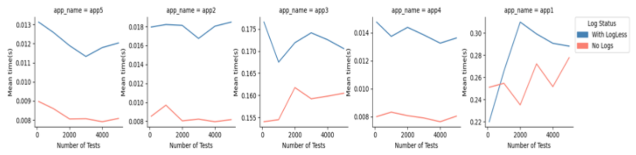


Figure 7: Line graph of mean time in seconds for apps with LogLess and No Logs

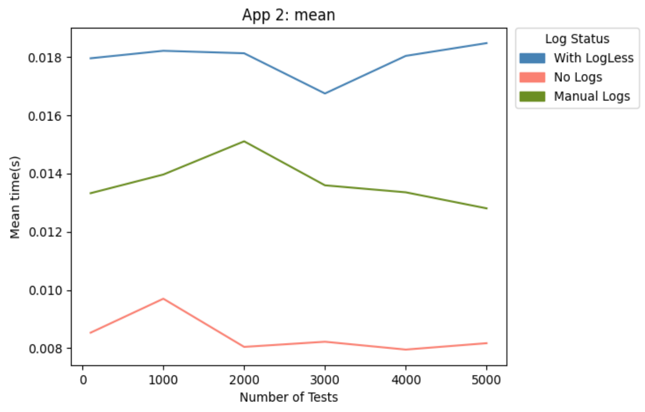


Figure 9: Mean time in seconds for LogLess, No Logs, and Manual Logs

B. *Unit Tests & Application Execution*

Another focus of experimentation was on accuracy, which is supported through unit testing and manual application testing. In order to validate the application, unit tests are provided, especially for the core functions. Additionally, code coverage was measured throughout development. The team seeks to continue to add tests and grow the code coverage. Similarly, the results of manual testing are verified through the human-in-the-loop methodology. The team inspected the results for correctness and the sample runs are shown in the LogLess GitHub repo.

VII. Related Work

The project's primary concern regarding software logging will be log instrumentation. This stage consists of including log statements in the application and maintaining them throughout the software development lifecycle. The logging approach, Logging Utility (LU) integration, and Logging Code (LC) composition are the steps of log instrumentation. The logging approach is the high-level workflow of how logging will be performed. The next component of LU integration involves deciding which tool to adopt (example: SLF4J for Java) and how to configure it. Finally, LC composition requires addressing where logging should occur, what should be logged, and how to log with high quality [7]. Documentation of these steps will be critical for this project.

Software logging with security practices is of paramount importance. To follow and implement best security practices when designing an application, vulnerabilities must be looked at. Insufficient logging is one example that can complicate bug identification and expose security vulnerabilities. The Open Web Application Security Project (OWASP) identified insufficient logging as one of the top ten security risks in 2017 [8,9]. In a serverless architecture, pinpointing bugs can become challenging if there are not enough log statements in the functions.

In addition, malicious attackers may have had attempts to steal sensitive information, but it could go unnoticed if the pertinent logging is absent [10]. For this project, understanding this vulnerability will be important to design the logging module; brainstorming the LC composition design effectively can help prevent or mitigate it. If our team plans to test this model in a distributed system consisting of multiple serverless functions, adapting some aspects of a shared log may be helpful. A team from the University of Texas have proposed a serverless runtime, Boki, that outputs a shared log API. This approach consists of building an ordered log that can be accessed concurrently, and is known to achieve scalability, strong consistency, and fault tolerance [11,12].

There has been extensive work being done in the application logging. We have conducted a comprehensive survey of works in this area, and identified a number of related works which in some way mark the stepping stones for the LogLess approach .

A. *Platform Agnostic*

Today most serverless platforms along with their logging and analysis tools are locked into a provider, for example, Lambda functions can only be used on AWS. Fear of provider lock-in has led to platforms such as *KNative*. KNative is an open-source, extensible, and flexible serverless platform built on top of Kubernetes. However, in the case of KNative more informative and fine-tuned logging requires other log services such as FluentBit to be configured [13]. Our tool offers a novel platform-agnostic logging tool that can provide informative and personalized logging to developers.

Diving into KNative more, this platform is managed by three core components: autoscaler, placement engine, and load balancer. In practice, development with KNative can consist of writing functions from a list of supported languages. These functions get deployed into worker pods, which comprise the queue proxy and function containers. The mentioned components will manage these worker pods throughout the lifecycle. Mittal and team leverage these KNative components to develop a resource management framework for the edge cloud environment [14].

B. Logging

Witt is a visual tool that takes serverless execution logs to present to the developer. With this, they present a timeline that explains the performance, structure, and data flow of a serverless function [15].

Alves and colleagues showcased that the benefits of logging can be evident through an examination of existing open-source repositories. Identifying the popularity of logging libraries can assist new development teams and researchers select one suitable for their projects. Another critical component of logging is surveying what type of information to log. This survey can be accomplished by extracting logger calls from existing repositories and persisting metadata associated with them, such as project name, context, verbosity, and message. In addition to this information, it can be beneficial to track the percentages for both logger calls and number of unique message words. Enumerating the list of top words for the different verbosity levels can benefit new development to adapt these descriptive logging words to offer a better level of standardization and log analysis [16].

Gu and team performed a comprehensive systematic mapping study (SMS) of logging practices. This study was an amalgamation of logging-based research topics, solution approaches, and research issues. It mainly identifies that most existing research has answered the "where" and "what" should be logged, but there are limited studies on answering "why" and "how" well to log. Improper usage of log levels, lack of crucial information, and low density of logging are further limitations that display the necessity for better logging guidance. A core recommendation from this study was that logging intentions and concerns should be followed; this includes examining contextual factors, performance overhead limits, and source. This study can supplement our research problem such that holistic considerations including the "why" and "how" questions of logging can be included when designing the logging module [17].

A project concern for software logging will be log instrumentation. This stage consists of including log statements in the application and maintaining them throughout the software development lifecycle. Logging approach, logging utility integration, and logging code composition are the steps of log instrumentation [7].

C. Code Modeling, Syntax, and Semantics

With various syntactic structures and user-defined functions successfully modeling code for our tool is challenging. Turning to prior works, Abstract syntax trees (AST) are the standard starting point for code modeling in code analysis. Furthermore, in-built AST conversion modules such as *ast* module in Python provides specific error messages as well as line location for syntactically invalid code [18]. Structural language modeling or SLM is an example of ASTs being used as a starting point. SLM takes AST as a base input and then applies decomposition to present partial trees in the form of paths [19].

In addition to this to capture code fully, semantics along with syntax must be taken into consideration. Code clone detection research completed by *Kalita* and *Sheneamer* [20] shows when adding both syntactic and semantic features to their model performance improves by 19.2% across all classification algorithms. CC-GGNN is an example that both takes AST as a starting point and leverages the importance of semantic features to create a code completion tool that outperforms most current state-of-the-art methods [21].

D. Machine Learning

Machine learning is commonly used for code classification, its usage can be noted in various tools that aid in code clone detection and review comments for example. Transformer-Based Code Classifier or TBCC is a proposed tool that uses deep learning classification over tree structures to accurately identify approximately 96% of C and Java code clones [22]. Research conducted by *Arafat*, *Sumbul*, and *Shamma* to categorize the quality of review comments also uses machine learning focusing on the semantics of the comments instead of the code which led to the models' poor performance [23].

More specifically natural language processing is utilized frequently in code analysis, industry examples of this include GitHub's Copilot. Copilot uses deep learning to provide the ability to autofill repetitive code, suggest test code, and convert comments to code [24]. Copilot is built using Codex which is a generative pre-training transformer language model made by OpenAI. In a paper released by OpenAI the authors note the benefits of using open-source code available on GitHub and code from programming competitions to train and assess their model [25].

Li and colleagues develop a model that recommends whether a given block of code should have a logging statement or not. The implementation consists of examining an AST of the source code and passing the features of it as arguments to the machine learning model. The team tested with a combination of syntactic and semantic features, and ultimately found that high usage of syntactic features resulted in a high balanced accuracy.

E. Security

Currently, there are only a few studies on security for serverless applications. To address this concern, Kim and colleagues have devised a set of guidelines and examples for designing a serverless environment in order to avoid security vulnerabilities. The team set up a set of services in AWS and performed a threat analysis using the STRIDE methodology. This analysis resulted in a set of common vulnerabilities that occur in serverless applications.

The insufficient logging vulnerability is one relevant example from the set that can make problem identification take a longer time and enable attacks to go unnoticed. This vulnerability is relevant to our project, and we can leverage the paper's suggestions to ensure our logging library writes important information that can fast track the surveillance of malicious attacks and monitoring process for analysis [10].

VIII. CONCLUSION

LogLess is one of a kind logging mechanism which not only reduces the manual logging efforts spent on serverless application development, but also provides a new approach to standardize logging. The novelty of LogLess comes from the fact that there hasn't been any such logging library out there which can efficiently write logs on the go. The tracing functionality of logless can very much be customized for any type of application code, irrespective of platform and language dependency.

The various features provided by logless can significantly take developer's efficiency to the next level as they can now focus only on the application logic without bothering about writing logging statements for debugging and audit purposes.

All work related to LogLess can be found at the following GitHub link: https://github.com/kushagrasoni/LogLess_Logging. This repo includes LogLess source code, test applications, units tests, benchmarking scripts, results tables, and graphs. LogLess will soon be available for public use at PyPi.

IX. References

[1] K. D. Smith, J. Jewett, S. Montanaro and A. Baxter, "PEP 318 – Decorators for Functions and Methods," June 2003. [Online]. Available: https://peps.python.org/pep-0318/.

[2] geeksforgeeks, "Python Inner Functions," November 2019. [Online]. Available: https://www.geeksforgeeks.org/python-inner-functions/.

[3] The Python Standard Library, "sys.settrace(tracefunc)," December 2022. [Online]. Available: https://docs.python.org/3/library/sys.html#sys.settrace.

[4] T. Popovic, "Advanced Python Techniques: Decorators," in *20th Conference on Information Technology IT '15*, 2015.

[5] The Python Standard LIbrary, "inspect — Inspect live objects," 2022. [Online]. Available: https://docs.python.org/3/library/inspect.html.

[6] The Python Standard Library , "ast — Abstract Syntax Trees," 2022. [Online]. Available: https://docs.python.org/3/library/ast.html.

[7] The Python Standard Library, "sqlite3 — DB-API 2.0 interface for SQLite databases," 2022. [Online]. Available: https://docs.python.org/3/library/sqlite3.html.

[8] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy and J. M. Patel, "SQLite: past, present, and future," *Proceedings of the VLDB Endowment,* vol. 15, no. 12, p. 3535–3547, 2022.

[9] M.-A. Lemburg, "PEP 249 – Python Database API Specification v2.0," 29 March 2001. [Online]. Available: https://peps.python.org/pep-0249/. [Accessed 2022].

[10] C. Boyuan and (. J. Zhen Ming, "A Survey of Software Log Instrumentation," *ACM Computing Surveys,* vol. 54, no. 4, pp. 1-34, 2022.

[11] F. Rivera-Ortiz and L. Pasquale, "Automated Modelling of Security Incidents to represent Logging Requirements in Software Systems," in *The 15th International Conference on Availability, Reliability and Security*, New York, 2020.

[12] R. G. Lennon and W. O'Meara, "Serverless Computing Security: Protecting Application Logic," in *2020 31st Irish Signals and Systems Conference (ISSC)*, Letterkenny, Ireland, 2020.

[13] Y. Kim, J. Koo and U.-m. Kim, "Vulnerabilities and Secure Coding for Serverless Applications on Cloud Computing," in *Human-Computer Interaction. User Experience and Behavior: Thematic Area, HCI 2022, Held as Part of the 24th HCI International Conference, HCII 2022*, Virtual Event, 2022.

[14] Z. Jia and E. Witchel, "Boki: Stateful Serverless Computing with Shared Logs," in *SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, New York, 2021.

[15] J. Manner, S. Kolb and G. Wirtz, "Troubleshooting Serverless functions: a combined monitoring and debugging approach," *SICS Software-Intensive Cyber-Physical Systems,* no. 34, pp. 99-104, 2019.

[16] N. Kaviani, D. Kalinin and M. Maximilien, "Towards Serverless as Commodity: a case of Knative," in *Fifth International Workshop on Serverless Computing*, Davis, CA, 2019.

[17] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. K. Ramakrishnan and T. Wood, "Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds," in *SoCC '21: Proceedings of the ACM Symposium on Cloud Computing*, Seattle, WA, USA, 2021.

[18] K. Shih-Ping Chang and S. Fink, "Visualizing Serverless Cloud Application Logs for Program Understanding," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.

[19] M. Alves and H. Paula, "Identifying Logging Practices in Open Source Python Containerized Application Projects," in *XXXV Brazilian Symposium on Software Engineering*, 2021.

[20] S. Gu, G. Rong, H. Zhang and H. Shen, "Logging Practices in Software Engineering: A Systematic Mapping Study," *IEEE Transactions on Software Engineering,* 2022.

[21] A. W. Wong, A. Salimi, S. Chowdhury and A. Hindle, "Syntax and Stack Overflow: A methodology for extracting a corpus of syntax errors and fixes," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[22] U. Alon, R. Sadaka, O. Levy and E. Yahav, "Structural Language Models of Code," in *International Conference on Machine Learning*, 2020.

[23] A. Sheneamer and J. Kalita, "Semantic Clone Detection Using Machine Learning," in *IEEE International Conference on Machine Learning and Applications*, 2016.

[24] K. Yang, H. Yu, G. Fan, X. Yang and Z. Huang, "A graph sequence neural architecture for code completion with semantic structure features," *Journal of Software: Evolution and Process,* vol. 34, no. 1, 2022.

[25] G. Liu and W. Hua, "Transformer-based networks over tree structures for code classification," *The International Journal of Research on Intelligent Systems for Real Life Complex Problems,* p. 8895–8909, 2022.

[26] Y. Arafat, S. Sumbul and H. Shamma, "Categorizing Code Review Comments Using Machine Learning," in *Proceedings of Sixth International Congress on Information and Communication Technology*, 2022.

[27] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," in *19th International Conference on Mining Software Repositories*, Pittsburgh, PA, 2022.

[28] J. T. H. J. Q. Y. H. P. d. O. P. J. K. H. E. Y. B. N. J. G. B. A. R. R. P. G. K. M. P. H. K. G. S. P. M. Mark Chen, "Evaluating Large Language Models Trained on Code," *CoRR,* vol. abs/2107.03374, 2021.

[29] Z. Li, T.-H. Chen and W. Shang, "Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks," in *IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[30] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu and D. Zhang, "Learning to Log: Helping Developers Make Informed Logging Decisions," in *IEEE International Conference on Software Engineering*, Florence, Italy, 2015.

[31] "Decorators for Functions and Methods," [Online]. Available: https://peps.python.org/pep-0318/.

[32] "Decorators for Functions and Methods," [Online]. Available: https://peps.python.org/pep-0318/.