LogLess:
A Logging Paradigm for Serverless Applications

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

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

Simranjit Bhamra
Masters 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. Ultimately, this idea 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 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.

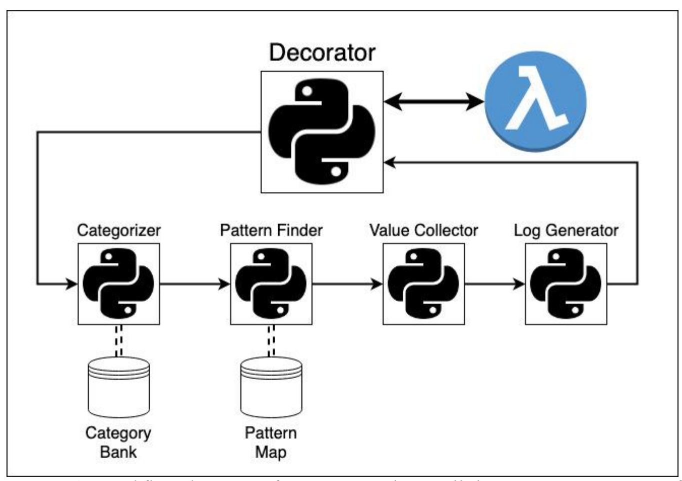

Figure 1: Workflow diagram of LogLess. It shows all the major components of the model

Many serverless platform providers have predefined logging of their cloud services. For example, in AWS, a

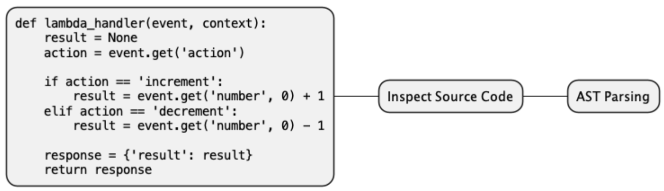

Figure 2.1: Inspecting the Source Code and Parsing using AST

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.

II. *The Decorator*

In most of the modern programming languages, like Python, Java, JavaScript, etc. everything is an object. The values, classes, instances, methods, functions, and other language constructs are objects. 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).

This is a parsing module, which parses the functions within the serverless deployed application. It traverses through every code-block within the function and passes them to the code categorizer [1].

A. Code Categorizer

- This module traverses through each code-block and categorizes the code according to the predefined categories. It utilizes *inspect* and *ast* modules of Python3 package. These categories will be looked up from a code category bank. This code category will then be passed to a pattern finder.

B. Code Category Bank

- This data bank will consist of various categories of code that might occur in a serverless architecture. This will categorize various programming functionalities like variable assignment, conditionals, loops, etc. Additionally, it will also contain categories defined based on extensive research on the available functionalities across various Serverless providers and their respective FaaS services.

C. Pattern Finder

- This module will determine the pattern of the categorized code and based on the category and pattern

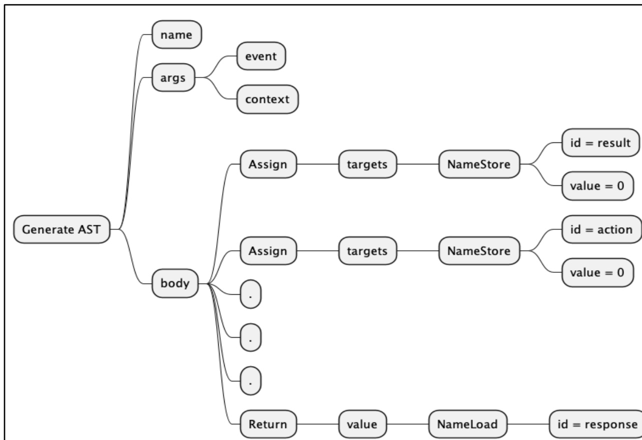

Figure 2.2: Generating Abstract Syntax Tree based on the application code.

map; it will determine the type of logging needed for that code category.

D. Pattern-Category Map

- This module will map the code category with its relevant logging pattern. This pattern will be created based on extensive research and experimentation for various types of applications running over serverless platforms.

E. Value Collector

- This module will store the values of all the variables occurring within a particular line/block of code.

F. Log Generator

- It takes the pattern and the variables and their corresponding values as an input, will generate an appropriate log, which will be descriptive and informative for developers and code reviewers.

III. *The Implementation*

We define a wrapper function which serves as a decorator for the function(s) in the application code. We named the decorator of our *LogLess* package – *decolog*.

A. Code Categorizer

To categorize every line of code of the decorated function, we first utilize the *inspect* module [2] of Python3 package to retrieve the *source-code* as text. Then, we parse the generated *source-code* to generate the Abstract Syntax Tree (AST) of the code in hierarchical structure. For this we are utilizing *ast* [3] which is also present as a python module and available as part of the standard python distribution.

The AST module helps Python applications to process trees of the Python abstract syntax grammar (Figure 2.1 & 2.2). It helps to find out programmatically what the current grammar looks like for the underlying python code.

Once the AST of the code is generated, we do a semantic analysis of the AST and categorize the various code blocks using Code Modeling, Syntax, and Semantics [*$VI.C*] along with a few Machine Learning models [*$VI.D*]. The categorization process utilizes a predefined set of rules and category database.

B. Code Category Bank

In order to assign a category to semantically categorize code for writing the logging, we created a light-weight database. This database comes in-built as part of the LogLess package and is based on *SQLite* [4, 5].

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. The *sqlite3* python module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by PEP 249 [6], and requires SQLite 3.7.15 or newer.

IV. EXPERIMENTATIONS

For experimentation, we plan to research a variety of applications currently prevalent on serverless platforms for their type of built-in functionalities. Based on that research we plan to create a data bank of all programming concepts and functions which are frequently used in serverless applications. We will also populate the categorization data bank with serverless utility-specific methods, and functions.

Once we have a data suite ready, we will have a robust and extensive set of unit tests to verify the functionality of logging and what is expected for each category of code and functions. The team plans to perform black box testing through equivalence partitioning and boundary value analysis. In addition, the team will utilize a code coverage tool to carry out white box testing. In testing the team plans to use tools such as *pytest* and *Coverage.py*.

V. LITERATURE SURVEY

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].

VI. RELATED WORK

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. Six code block categories are identified for classification through the model, such as catch, branch, and iteration. This paper addresses the limitation of limited logging recommendations in existing studies, which can benefit our project in the selection of a ML model for the pattern finder component [26].

Zhu and team explore and develop a tool, LogAdvisor, that provides actionable suggestions on where logging statements should be inserted. This implementation focuses on strategic logging placement to capture significant runtime information. With this intention, it is also worth noting that unintended consequences should be avoided, such that minimal logging can circumvent important information and excessive logging can both increase resource consumption and cause information masking. The machine learning workflow consists of instance collection, label identification, feature selection, model construction, and logging suggestion. Our project involves adding critical logging statements for users, so leveraging ideas from the LogAdvisor tool can benefit us in pinpointing logging locations [27].

Our tool plans to utilize machine learning techniques to aid in the classification of code and provide helpful logging to developers. Furthermore, as far as our research shows we are the first to use this to provide an open source serverless logging platform.

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].

VII. REFERENCES

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

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.