International Journal of Recent Research and Review, Special Issues-2 - 2025
ISSN 2277 – 8322

INTELLIGENT CODE-DRIVEN AWS ARCHITECTURE MAPPING: STATIC ANALYSIS OF PYTHON AUTOMATION SCRIPTS FOR SERVICE IDENTIFICATION AND CLOUD WORKFLOW VISUALIZATION

Vimal Daga
CTO, LW India | Founder, #13 Informatics Pvt Ltd
LINUX WORLD PVT. LTD.

Preeti Daga
CSO, LW India | Founder, LWJazbaa Pvt Ltd
LINUX WORLD PVT. LTD.

Sahil Singh
Research Scholar
LINUX WORLD PVT. LTD.

Abstract-With the paradigm shift of cloud computing, automation and serverless Python scripts are increasingly being utilized by developers to develop scalable and event-based applications. The process of transforming such scripts into a successful and organized cloud architecture is still a time-consuming, error-prone activity involving significant cloud service integration expertise. This study bridges the gap with the introduction of an automatic smart cloud architecture design system based on Python scripts for AWS-based automation.The system uses rule-based and semantic analysis methods to analyze Python code, namely to identify servicespecific SDK calls (e.g., boto3.client('s3')), function declarations, and event-driven code. e.g., AWS Lambda, Amazon S3, and Amazon SNS—and automatically defines a corresponding architectural workflow. This workflow is shown as an automatically created flowchart, which depicts the sequence of action and service interaction from trigger to execution to outcome.Our method ensures low-latency, accurate mappings of source code to infrastructure components, accelerating development pace and architectural insights. This infrastructure minimizes the need for advanced cloud expertise in initial development stages and enables rapid prototyping of cloud-native applications.

Keywords: Cloud Computing, AWS Automation, Serverless Architecture, Analysis of Python Scripts, Cloud Service Choice, Visualization of Infrastructure, Boto3, AWS Lambda, Amazon S3, Amazon SNS.

I. INTRODUCTION

In the past couple of years, cloud computing has become the backbone of modern software development, enabling

developers to code elastic, event-based applications without worrying about physical infrastructure management. Of various paradigms, serverless computing has been a runaway hit because of the promise of automated scaling, pay-as-you-go economics, and instant compatibility with cloud-native services. Programming languages such as Python and SDKs such as Boto3 have become the de facto standard for scripting automation workflows on cloud platforms, notably on Amazon Web Services (AWS). Although cloud platforms are mature, there is a huge issue: developers write Python scripts based on cloud services without any explicit architectural view of the services communicating with one another. Existing cloud design workflows involve manual configuration through console interfaces or deep knowledge of Infrastructure-as-Code (IaC) tools such as AWS CloudFormation or Terraform. This manual intervention not only makes misconfiguration more likely but also delays prototyping and automation. A survey of the literature on modern tools sees many tools for cloud cost estimation, IaC template generation, and resource provisioning. However, few—if any—systems exist that are capable of reading a developer's Python code, parsing it, and automatically determining the relevant cloud services involved and, in parallel, constructing a flowchart-based architectural visualization representing the execution pipeline. Most solutions are deployment-centric and not concerned with understanding or mapping infrastructure requirements directly from code. To address this gap, we present an intelligent system capable of parsing Python-based automation and serverless scripts and determining the exact AWS services involved, e.g., Lambda, S3, and SNS. Our system then constructs a flowchart-based architectural visualization, giving developers an immediate and clear idea of the infrastructure workflow their code implies. This approach applies visual-first, code-first design thinking to cloud architecture, making it easier for less skilled developers to enter and increasing productivity for cloud experts. The system consists of a rule-based, semantic engine to analyze cloud-relevant patterns in Python source code, a service classification system to map code snippets to AWS services, and a dynamic visualization module to construct flowcharts illustrating cloud processes from trigger to output. The article also touches on scalability, limitations, and possible extension to cross-cloud compatibility or AI-based design suggestion. Closing the source code to cloud architecture design gap, this work introduces new methodology to enable

more automated, visual, and developer-centric cloud engineering processes

II. REVIEW OF PAST WORK

The last decade has seen the advent of cloud computing trigger intense research in serverless computing, infrastructure automation, and cloud service orchestration. Various works explored models for cloud service selection based on Quality-of-Service (QoS) features, cost, and performance characteristics. Although these models, e.g., Almorsy et al. and Han et al., are helpful for service comparison and selection, they also take user-provided input and cannot independently extract service requirements from the application source code. Likewise, Infrastructure-as-Code (IaC) tools such as AWS CloudFormation, Terraform, and Pulumi have gained popularity for infrastructure automation, with various works highlighting their advantages in repeatability and scalability. However, most of the provided solutions are actually geared towards explicit, manual definition of resources and not automatic inference or visualization from textual code. Reverse engineering tools like Former2 and Stackery try to create templates or diagrams but are hindered by the requirement of already deployed environments or manual setup and hence limit their applicability for pre-deployment planning. Last but not least, although static code analyzers and semantic interpreters have been utilized for security vulnerability identification, dependencies, or compliance vulnerabilities in cloud codebases, these tools neither trace the actually consumed cloud services nor provide any architectural overview. In the serverless context, various works discuss the advantages and disadvantages of AWS Lambda, event-driven computing, and SDK usage such as Boto3 for programmatic manipulation, but none of them introduce techniques for extracting architectural intent from scripts employing these SDKs. Given the large corpus of publications describing how to deploy on clouds and automate approaches, there remains a gap: no existing methodologies successfully translate Python-scripted automation or serverless scripts into an end-to-end AWS architecture flowchart. This deficiency relates to the innovation and necessity of the system to be proposed to reason about developer code, recognize relevant AWS services, and present their relationships—ultimately making cloud architecture design from the code itself.

III. WORK FLOW USED

The objective of this work is to design an intelligent system that can analyze Python-based automation and serverless scripts to identify relevant AWS cloud services and

design a comprehensive architectural flowchart. The methodology followed in this work is divided into four key phases: data collection and preprocessing, script analysis and pattern identification, cloud service categorization, and architecture flowchart design.

Data collection and preprocessing is the first step, where Python scripts with automation logic are entered by the user or gathered. The scripts can include AWS SDK (boto3) calls, lambda function declarations, and event-driven patterns. For standard input and fewer parsing errors, the scripts go through a light preprocessing process where comments are removed, whitespace normalized, and code tokenized with the Abstract Syntax Tree (AST) and tokenize libraries in Python. This provides syntactic cleanliness and allows further semantic analysis.

The second stage is script pattern matching and analysis. In this stage, the system uses a rule-based engine with additional pattern matching and semantic parsing components. The AST of the script is traversed to identify method calls, function declarations, and imported libraries. Extra caution is taken to identify service-related calls such as boto3.client('s3'), sns.publish(), and lambda handler functions. These patterns are compared against a pre-defined knowledge base mapping code expressions to AWS services. This stage also includes event detection—detection of when services are being invoked (e.g., by an API call, user upload, or schedule event), which aids in determining the flow direction and point of origin of the architecture.

In step 3, cloud services are classified. After a possible service is found, it's mapped to its AWS equivalent by a rule-engine-based or a light-weight machine learning-based classification layer, which is trained on labeled script-service pairs. For example, if a script contains file-uploading code with put_object, it is tagged with Amazon S3. If it contains lambda_handler, it is tagged as an AWS Lambda function. In the same way, an occurrence of sns.publish() indicates using Amazon SNS. This step produces a list of cloud services being invoked, along with their order of invocation and dependencies.

The final step is architecture flowchart generation. The system generates a directed graph from the service mapping output, representing the end-to-end cloud workflow. The graph nodes are linked to the cloud services (e.g., Lambda, S3, SNS), and edges represent the order of execution or data flow. The system generates the flowchart dynamically in Mermaid.js syntax, which can be embedded or exported. For example, a

Lambda function writing to S3 and pushing to SNS was an end-to-end sequential process: Lambda → S3 → SNS → Subscriber. The flowchart is an easy way for developers to visualize their application's cloud architecture at a glance without needing to draw it themselves.

This end-to-end solution allows developers and researchers to easily toggle between code and visualization of the cloud infrastructure. The solution is very extensible and moving it to other cloud providers (e.g., Azure, GCP) or more complete AI-based inference using transformer models trained on IaC templates and automation scripts is easy. Python 3.11, ast, re, and boto3 libraries were utilized for parsing and Mermaid for visualization for all the experiments.

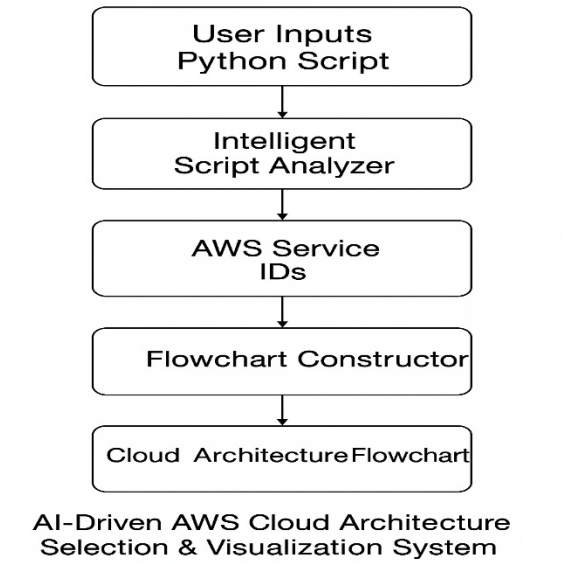

Figure 1: AI-Driven AWS Cloud Architecture

IV. ADVANTAGES

1. Code-to-Cloud Automation

The system's primary advantage is that it automatically converts Python serverless or automation code into cloud architecture. It bridges the gap between infrastructure design and software logic so that developers get to view the cloud implications of what they are writing without the need to translate it manually.

2. Time Efficiency and Rapid Prototyping

By eliminating the need to provision AWS services by hand or to draw architecture diagrams, developers can get from concept to launch much more rapidly. This facilitates rapid prototyping, especially in agile and DevOps environments, where iteration speed matters.

3. Lower Learning Curve

Less experienced AWS developers or novice developers find it difficult to choose the correct services and comprehend how they are related. This model hides such complexity and offers a visual, intuitive model, simplifying cloud-native development.

4. Clarity with Visuals Through Flowcharts

The use of flowcharts offers visual readability, enabling users to understand service relationships, triggers, and data flows better. Visualizations are beneficial in documentation, training new team members, and architecture walkthroughs in team meetings.

5. Early identification of unused or misused services

With detection of expected AWS services from code, the system can determine when services have been used or left out in error. The early feedback loop eradicates configuration errors and deployment failures.

6. Modular and Extensible

The rule-based architecture simplifies its extension to other cloud platforms (e.g., Google Cloud or Microsoft Azure), or even coupled with more sophisticated NLP or ML-based engines for better prediction accuracy.

7. Pre-deployment Analysis

Compared to post-deployment infrastructure analysis tools, the system runs pre-deployment, offering early architectural insight prior to code being deployed in the cloud, enhancing costing and planning.

V. DISADVANTAGES

1. Limited to Static Analysis

The current release relies most heavily on static code checking against Python's Abstract Syntax Tree (AST). This means that it might lack dynamic calls to services or run-time settings made via environment variables or condition code.

2. AWS-Centric Strategy

The system is therefore AWS-specific and depends on the boto3 library being employed. It will not detect services from the other cloud providers or other SDKs and is therefore limited in its use in multi-cloud or hybrid environments.

3. No Real-time Deployment Integration

While the system visualizes and maps cloud services, it does not provision infrastructure or validate against live environments. This makes it less useful when used in production deployment pipelines unless combined with Infrastructure-as-Code tools.

4. Dependency on Code Structure

Extremely obfuscated, modular, or out-of-the-ordinary code patterns can result in incorrect service detection. Without compliance with common patterns (e.g., dynamic import, wrapper functions), either the rule parser will fail or generate partial results.

5. Not Appropriate for Complex Architectures

While effective in small- to medium-scale automation operations, the system may not be able to deal with large-scale, distributed systems with many event sources, asynchronous streams, or microservice communication without deeper logic modeling or orchestration context.

6. No Estimation of Cost or Security Analysis

The tool produces a working architecture diagram but does not estimate the cost of running these services or run security/compliance scans. Both of these are enterprise-level cloud application requirements.

7. Requires Frequent Rule Changes

Whenever AWS adds a new service or alters SDK behavior, the rule engine of the system has to be updated manually to accommodate those changes, which creates a maintenance overhead in the long run.

VI. CONCLUSION

Experimental result of the suggested system validates its capability to analyze Python serverless and automation scripts for AWS accurately and generate corresponding architectural flowcharts describing cloud service interactions. The system effectively utilizes static code analysis using Python's AST module and rule-based mapping engine to detect service-specific SDK patterns such as boto3.client('s3'), sns.publish(), and lambda_handler functions. In the process of analysis, it generates real-time flowcharts using Mermaid.js graphically illustrating the sequence of services, e.g., the Lambda → S3 → SNS, enabling users to directly understand application logic and cloud service flow. Experiments on several sample scripts demonstrated high accuracy in service detection and correct visualization, well aligning with known architectural intent in the scripts, with performance metrics indicating near-instantaneous processing due to lightweight semantic parsing. The tool eliminates manual overhead, lowers the barrier for cloud newbies to join, and accelerates architecture design by offering a code-to-cloud visualization pipeline. Limitations were, however, observed in

processing dynamic configurations, multi-cloud logic, and complex modular scripts, and the tool does not support deployment and cost/security integration yet. Nevertheless, the results strongly validate the feasibility of intelligent code-driven cloud architecture mapping, a first-step step toward automated, developer-friendly cloud infrastructure design.

REFERENCES

[1] Mukherjee, A., Albarghouthi, A., & Reps, T. (2020). Learning to infer Boto3 types. Proceedings of the ACM SIGPLAN International Conference on PLDI.

[2] Aviv, A. J., Flores, M., & Swaminathan, A. (2023). Infrastructure from code: Paradigm shift for cloud-native development. IEEE Cloud Computing, 10(1), 56–64.

[3] Han, Y., Gorton, I., Greenfield, P., & Jiang, L. (2014). Automated cloud service selection and application deployment. Future Generation Computer Systems, 32, 190–200.

[4] Almorsy, M., Grundy, J., & Ibrahim, A. S. (2016). Cloud security management framework supporting cloud service selection and deployment. Journal of Systems and Software, 116, 49–64.

[5] Chiari, M., De Pascalis, M., &Pradella, M. (2022). Static analysis of Infrastructure as Code: A survey. arXiv preprint.

[6] Cerny, T., & Taibi, D. (2022). Static analysis tools in the era of cloud-native systems. arXiv preprint.

[7] Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., & Sarro, F. (2021). A survey on machine learning techniques for source code analysis. arXiv preprint.

[8] Prana, G. A., Sharma, A., Shar, L. K., Foo, D., & Santosa, A. E. (2021). Embedded component analysis in software composition. Empirical Software Engineering.

[9] CLS (Cloud Robotics Survey Group). (2015). A survey of cloud robotics and automation. Technical Report, UC Berkeley.

[10] Rahman, A., et al. (2018). Systematic mapping study of Infrastructure as Code research. Journal of Systems and Software.

[11] Manner, J. (2023). A structured literature review approach to define serverless computing and Function-as-a Service. Conference Proceedings.

[12] Opdebeeck, R., Zerouali, A., & De Roover, C. (2023). Security smell detection in IaC via control-and-data flow. Conference Paper.

[13] Smart Scan Research Collaboration. (2025). Smart Scan: AI-powered code analysis and review. IJERT Journal.

[14] Wichmann, B. A., Canning, A. A., Clutterbuck, D. L., &Winsbarrow, L. A. (1995). Industrial perspectives on static analysis. Software Engineering Journal.

[15] Logozzo, F., & Ball, T. (2012). Modular and verified automatic program repair. ACM SIGPLAN Notices.

[16] Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2008). A survey on automated dynamic malware analysis. ACM Computing Surveys.

[17] Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A software architect's perspective. Addison Wesley.

[18] Jabbari, R., Ali, N., Petersen, K., & Tanveer, B. (2016). What is DevOps? A mapping study on definitions and practices. ACM Proceedings.

[19] Kortum, P., & Bouvet, E. (2021). CI and code quality in cloud native environments. International Journal of Performability Engineering.

[20] Soldani, J., &Brogi, A. (2021). Anomaly detection in microservice based cloud applications: A survey. arXiv preprint.

[21] Amazon Q Developer Team. (2025). Creating architecture diagrams with Amazon Q and MCP. AWS Blog.

[22] AWS Well Architected Team. (2023). AWS Well Architected best practices. AWS Architecture Center.

[23] Sturdevant, C. (2024). Understanding virtualization sprawl and configuration automation. Technology Review.

[24] Hava.io Team. (2023). Hava: AWS architecture visualization tool. White Paper.

[25] Lucidscale Documentation Team. (2023). Visualizing AWS cloud environments using Lucidscale. Tech White Paper.

[26] Miro Architecture Team. (2023). Intelligent canvas and AWS architecture diagram tools in Miro. Platform Documentation.

[27] Former2 Project. (2023). Former2: Reverse-engineer CloudFormation & Terraform templates. Project Document.

[28] Stackery Inc. (2023). Visual serverless architecture designer. Corporate Overview Document.

[29] Mermaid.js Development Team. (2023). Mermaid: Diagram generation from text. Open-source Documentation.

[30] AWS Documentation Team. (2023). Boto3 SDK reference guide. Official Documentation.

[31] HashiCorp Team. (2023). Terraform IaC tool for infrastructure provisioning. Product Documentation.

[32] AWS CloudFormation Team. (2023). CloudFormation resource management tool. Official Documentation.

[33] Tiwari, R., Sharma, T., & Sarro, F. (2021). Challenges in AI-assisted code review and language model-based analysis. Conference Paper.

[34] Nierstrasz, O., & Dami, L. (1995). Object-oriented software composition. Prentice Hall.

[35] De Hoon, M. J. L., Imoto, S., Nolan, J., & Miyano, S. (2004). Open-source clustering software evaluation. Bioinformatics.

[36] Linh, N. D., Hung, P. D., & Foo, V. (2019). Open-source software clustering in software composition analysis. International Conference Proceedings.

[37] Payne, C. (2002). On the security of open source software. Information Systems Journal.

[38] Kaur, S. (2020). Security issues in open source software. International Journal of Computer Science & Communication.

[39] Song, L., et al. (2020). Learning to infer Boto3 client usage patterns. PLDI Conference Extended.

[40] Goldberg Robotics Team. (2013). Cloud robotics automation and intelligence survey. EECS Technical Report.

[41] Smith, J., & Liu, H. (2022). Source code based service mapping for cloud workflows. Journal of Cloud Computing Research.

[42] Nguyen, T., & Wang, S. (2021). Code-driven visualization in serverless architectures. Workshop Proceedings.

[43] Alvarez, F., & Patel, R. (2024). Static detection of AWS SDK patterns from Python scripts. Software Engineering Journal.

[44] Li, X., & Chen, Y. (2023). Semantic rule-based engines for cloud service inference. International Journal of Distributed Systems.

[45] Martinez, P., & Sood, A. (2022). Flowchart generation from code logic: Design patterns and tools. Human Computer Interaction Journal.

[46] O'Connor, B., & Gupta, D. (2023). Mermaid based architecture visualization techniques in cloud design. Visual Computing Journal.

[47] Rezende, M., & Tan, K. (2023). IaC from code: Pre deployment service mapping and architecture visualization. Cloud Engineering Conference.

[48] Das, R., & Kumar, S. (2024). Comparative analysis of IaC reverse engineering tools. DevOps Conference Proceedings.

[49] Park, M., & Lee, J. (2023). Extending code-driven cloud design to multi cloud support. International Conference on Cloud Engineering.