|  |
| --- |
| University Of Cincinnati  School of Information Technology (SoIT) |
| Comparison of terraform state files from Automated IaC deployment |
|  |

|  |
| --- |
| Kandukuri Sukeerthi |

Table of Contents

[Abstract 2](#_Toc164380491)

[1. Introduction 3](#_Toc164380492)

[2. Literature Review 4](#_Toc164380493)

[2.1 Introduction 4](#_Toc164380494)

[2.2 Related Works 4](#_Toc164380495)

[2.3 Conclusion 6](#_Toc164380496)

[3. Methodology 6](#_Toc164380497)

[3.1 Research Design 7](#_Toc164380498)

[3.2 Application Development Approach 7](#_Toc164380499)

[3.3 Requirement Analysis 8](#_Toc164380500)

[3.4 Design Phase 8](#_Toc164380501)

[3.5 Development 11](#_Toc164380502)

[3.6 Testing and Quality Assurance 11](#_Toc164380503)

[3.7 Iteration and Updates 11](#_Toc164380504)

[3.8 Evaluation Metrics 12](#_Toc164380505)

[3.9 Ethical Considerations 12](#_Toc164380506)

[3.10 Conclusion 12](#_Toc164380507)

[4. Results and Discussions 13](#_Toc164380508)

[5. Conclusions and Recommendation 15](#_Toc164380509)

[6. References 16](#_Toc164380510)

# Abstract

In the world of Infrastructure-as-Code (IaC), which is constantly developing, detecting infrastructure drift that poses significant challenges such as loss of control by inconsistency behavior, deployment errors impacting the application performance and availability and compliance deviations in environments that change dynamically after deployment. Therefore, the detection of infrastructural drift is essential as IaC evolves. This study aims to determine how to detect infrastructure drift in IaC deployments using Terraform files. In order to accomplish this purpose, an experimental and applied research method is used, and a comparison tool is built. This tool, crafted using Python, facilitates a comprehensive analysis by comparing Terraform state files with expected outcomes, presenting the findings through graphical representations. The tool's contributions are manifold: it enables easy identification of changes, reduces manual intervention by automating drift detection, and enhances understanding of deployment status, thereby fostering better consistency in infrastructure management. It provides a methodical way to reduce the risks related to infrastructure drifts, which makes it extremely important in the field of infrastructure management. The tool helps to increase consistency in IaC deployment by increasing system efficiency and dependability.

Keywords: Infrastructure as Code, Terraform, IaC, Infrastructure drift

# 1. Introduction

Imagine deploying a critical application to the cloud, only to have it fail mysteriously hours later. Unfortunately, this scenario is a common consequence of infrastructure drift. As the software industry moves at such a fast pace, development cycles are being shortened leading Infrastructure as Code (IaC) as an essential infrastructure management approach [1],[2]. Infrastructure as Code facilitates system configuration and deployment provisioning through code, utilizing Terraform, one of the most popular tools for infrastructure automation [3]. In addition, IaC provides cloud providers that further benefit Cloud Computing systems by enabling faster deployments, improved repeatability and maintenance simplification/updates. Nevertheless, any failure in scripting may lead to deployment drifts [4]. Also taking its toll on this problem by practices that were agreed upon globally by software engineers including using version control systems, automated testing and code reviews [5]. The inconsistently deployed infrastructure across different environments can produce systems failures, unexpected behavior or compatibility issues when. Thus, maintaining consistency in IaC throughout the process will guarantee predictability and dependability leading to reproducibility [6].

Despite various investigations carried out on ways of improving the deployment consistency. Existing solutions for managing IaC primarily focus on pre-deployment checks, ensuring code quality and using advanced modeling techniques. However, a crucial gap exists in post-deployment verification [7]. By focusing on the post-deployment phase, this study aims to bridge the gap left by existing solutions. This bridge matters because deploying drift can result in several grave concerns such as compliance issues; performance degradation and security vulnerabilities [8]. The key contribution of this research lies in its ability to identify inconsistencies after infrastructure is deployed. The present study tries to tackle this problem by coming up with a comparison tool specifically meant for evaluating the configuration status of public cloud infrastructure deployed using Terraform scripts. The study seeks to

1. How can comparison of a terraform state file be used to identify infrastructure configuration drift in automated deployments?

The methodology used for this project is software development with an emphasis on coming up with a way of determining infrastructure drift. The idea is to introduce a program that would compare state files against already created test cases. This uses graphical representation methods to facilitate easy understanding of the existing infrastructure, hence being able to appraise if there are any differences. Thus, when the application systematically examines the state of the infrastructure against anticipated configurations, stakeholders are enabled to identify and resolve instances of drift effectively [9]. This technique offers an efficient way of ensuring consistency in automated deployment infrastructures and therefore has potential for industry use. Essentially, it is no exaggeration that in view of today’s fast changing technology and digital revolution, this solution fits well in the industry and meets an urgent need for reliable infrastructure management.

Below paper sections are as follows: Section 2 explains Literature Review about Existing Solutions and Their Drawbacks; Section 3 Describes Approach Used for This Project; Section 4 Gives Findings and Analysis. Lastly, Section 5 Conclusion and Future Work.

# 2. Literature Review

## 2.1 Introduction

Although IaC has several advantages, there is a chance of deployer drift. This occurs when the deployed infrastructure deviates from its intended setup over time. Issues such as poor performance, security threats, and difficulty adhering to regulations may result from this. Numerous research investigations have attempted to address the complexity of regulating IAC configurations and minimizing the influence of drifts to address this. This section provides a thorough analysis of the literature with an emphasis on techniques, concepts, and inventions that improve the accuracy, speed, and ease of recovery of IaC procedures.

## 2.2 Related Works

Nemania Borovits [10] tackled the important problem of identifying linguistic anti-styles within infrastructure as code (IaC) scripts, which are essential for controlling and providing computer infrastructures. The problem's importance stems from the capability discrepancies between the names and the common body of IaC code units. This work used a unique automated method that leverages word embeddings and deep learning techniques to address linguistic difficulties in IaC scripts, whereas existing literature focuses mostly on the structural aspects of flawed IaC scripts. The methodology is divided into five stages, which include the following: corpus tokenization, data sets generation, from datasets to vectors, model training, and identification of inconsistencies. The technique produced accuracy values ranging from 0.785 to 0.915, an AUC metric from 0.779 to 0.914, and an MCC metric from 0.570 to 0.830, indicating promising accuracy, according to the data. Most remarkably, the method achieves an accuracy of 0.915 in detecting discrepancies inside file modules, indicating superior overall performance. Still, one limitation of the analysis is the very small number of repositories used for data processing. Despite this issue, the paper's value resides in its ability to provide users with automated support for troubleshooting discrepancies in both the names and contents of IaC code units. The developed solution's exclusion to the Ansible language creates an amazing gap that may be filled by future research to increase its applicability to other IaC systems.

The necessity to identify effective testing techniques to improve the quality of Infrastructure as Code (IaC) scripts was discussed in Hasan's work [11]. This issue is important because IaC scripts are prone to errors that might have serious consequences. This is proven by the $150 million USD outage that Amazon Web Services (AWS) suffered in 2017 because of an IaC script error. Internet artifacts, including blog posts and videos, are systematically analyzed in three stages as part of the research methodology used in this paper: Internet Artifact Collection, Internet Artifact Filtering, and Open Coding. Six crucial testing procedures for IaC scripts are identified because of the observation: using automation, sandbox testing, testing every IaC change, behavior-centered test coverage, avoiding coding anti-styles, and remote testing. Interestingly, developers are increasingly favoring the approaches of Testing Every IaC Change and Using Automation. The focus of the study is on providing actionable instructions for improving script quality, even though it offers practitioners valuable insights for guidance on IaC testing. Nonetheless, rater bias affects the derivation technique of the detected practices, which is a serious problem.  Additionally, there is still a gap in the study's dependence on online artifacts, which likely means that valuable insights from practitioner interviews are missed. Nonetheless, fundamental queries remain regarding the scope and relevance of the acknowledged practices.

Similar to [11], Rahmans research [12] focused on enhancing the precision of scripts through the development of a defect classification system specifically tailored for such scripts. The main goal of the study is to aid professionals in comprehending types of defects and identifying strategies for mitigating them. The methodology employed includes qualitative analysis examining 80,425 contributions from 291 Open-Source Software (OSS) repositories spanning from 2005 to 2019 to assess the prevalence of identified defect categories. The key findings indicate that issues related to idempotency are less common while defects associated with configuration settings are more prominent. Despite utilizing a set of keywords linked to disorders from studies there are limitations in this research. For instance, practitioners may still use terms that were not considered in the evaluation. Additionally, the empirical analysis focuses on Puppet, one IaC tool, which raises concerns about its generalizability. Nevertheless, this study holds significance as it presents a taxonomy of defects that could assist professionals in prioritizing their efforts towards verifying and validating their findings, managing defect resolution processes efficiently and determining which IaC scripts stand out as exemplary.

In contrast, to the studies mentioned earlier Daniels research [13] explores the drawbacks of static Infrastructure as Code (IaC) solutions, which are typically designed for deploying infrastructures that remain unchanged post deployment. The study underscores the importance of managing infrastructures that evolve over time requiring updates and modifications. The study methodology involves implementing a testing process using asset-based testing and environment simulation techniques for modern IaC applications. A key finding of the study is the implementation and evaluation of an extension built on Pulumi with Hareactive showcasing performance metrics similar to existing solutions. However, due to the complexity of specifying target configurations the analysis recognizes the challenge of ensuring accuracy in dynamic IaC systems. Notably the assessment of the proposed approach confirms its applicability to distributed processes. Despite its advantages, the research is confined to utilizing Pulumi TypeScript for static IaC implementation limiting its adaptability to frameworks and languages. In conclusion the study offers insights into advancing Infrastructure as Code solutions representing a promising advancement in this domain.

A tool created to address the challenge of preventing infrastructure discrepancies, within Opscodes Chef, a top-notch Infrastructure as Code (IaC) platform was developed by Hummer and his team [14]. Featuring a user interface, this tool—specifically tailored for Chef automation scripts—utilizes a version-based testing approach. Generates an Excel document that compares test scenarios and detects infrastructure modifications. The key insights gathered from the study reveal that the tool offers engineers easily to find the root causes of errors and resolve issues associated with discrepancies. While it plays a role in bolstering trial processes and drift prevention measures certain limitations of the tool are evident—it predominantly focuses on Chef scripts sourced online potentially restricting its applicability. Additionally, its exclusive compatibility with Chef scripts underscores a gap in its adaptability to frameworks suggesting opportunities for future research and progress in automated validation solutions, for more comprehensive IaC initiatives.

## 2.3 Conclusion

The five case studies offer insights into aspects of Infrastructure as Code (IaC) improvement and testing. They also highlight areas for research and development providing advice. Nemania Borovitss [10] analysis addresses the issue of identifying patterns in IaC scripts through a specialized automated method that utilizes deep learning and word embeddings. Hasans [11] research underscores the significance of testing procedures for scripts by identifying key testing practices. Rahmans [12] contributions expand knowledge on IaC defect taxonomy assisting practitioners in enhancing script quality and addressing issues. Daniels [13] analysis introduces solutions to overcome limitations of static techniques showing promising results in real world scenarios. While Hummer [14] and colleagues' tool based overall strategy for Chef automation scripts has limitations in precision and flexibility it can help prevent infrastructure drifts. These studies underscore the importance of development, testing and management processes to ensure consistency in IaC deployments while suggesting avenues for innovation and advancement in the field.

# 3. Methodology

This study introduces a comparison tool designed to visually depict infrastructure drifts post-deployment. It operates by comparing Terraform state files, which contain configuration details generated from automated Infrastructure as Code (IaC) deployments, against predefined test cases within the tool. The output of the tool includes two bar graphs, illustrating the variance between the expected and deployed infrastructure configurations.

## 3.1 Research Design

This study develops a research design that integrates both applied and experimental research approaches. With this method, the tool's design and implementation are done in addition to an assessment of how successfully it identifies and represents drift changes. While the applied research phase focuses on the tool's applicability and utility in real-world circumstances, the performance and efficacy of the tool's testing component are assessed against specified test cases.  This study approach guarantees a thorough grasp of the tool's capabilities and limits as well as any potential effects on infrastructure management techniques.

## 3.2 Application Development Approach

For enhancing the software development process the Fig1 Waterfall approach [15] was chosen for this project. This methodology is characterized by its progression through stages such as requirements gathering, system design, implementation, deployment and testing in a sequential manner. The clear and consistent project requirements supported this approach. Enabled a development procedure. Adherence to the Waterfall model ensured resource management. The model presented a framework for creating a product that met the project's needs through a systematic and disciplined development approach despite not having the flexibility of Agile methods.

A diagram of a software development process

Description automatically generated

Fig 1: Waterfall Model

## 3.3 Requirement Analysis

Based on a thorough analysis of several papers and sources pertaining to the deployment procedures, several key requirements were identified for the software tool:

Supported Cloud Platforms: The tool should be compatible with major public cloud platforms (e.g., AWS, Azure, GCP) to offer flexibility for users.

Terraform Integration: Seamless integration with Terraform is essential, enabling the tool to process and compare Terraform state files.

State File Comparison: The core functionality is to effectively compare deployed infrastructure against desired configurations, likely leveraging Terraform state files.

Graphical Representation: To simplify drift identification for users, the tool should present comparison results in an easily understandable visual format (e.g., charts, diagrams).

JSON Parsing: As Terraform configurations often use JSON format, the tool needs to possess capabilities to parse and analyze JSON data effectively.

## 3.4 Design Phase

The project framework's structure layout is described in Figure 2. First a repository is pulled from GitHub into the Azure devops and sent to the Azure pipelines, where continuous integration and delivery (CI/CD) is carried out. Custom application files are added to the pipeline along with Terraform scripts created by Azure. To implement a custom file in this research, Asp.Net files are included. .Net web applications are used together with App plan services that include app services to aid in the administration, scalability, and security of web applications. Following continuous delivery, a resource group with a virtual machine hosting the web application and blob storage to hold the Terraform state files is built. Simultaneously Python scripts are built for the comparison tool, which has a UI where the user compares the state file that is generated with the stated test cases to produce bar graphs.

A diagram of a computer system

Description automatically generated

Fig 2: Project Architecture Design

The High-Fidelity User Interface prototypes are shown in Figs. 3 and 4. The user can submit two files in Figure 3. Next, using the compare button leads you to Fig. 4, the results page. Figure 4 lists the deployed resources that were taken from the state file and shows two graphs that compare the predicted and actual results.

A screenshot of a computer

Description automatically generated

Fig 3: High Fidelity of Application’s Home Page

A screenshot of a computer screen

Description automatically generated

Fig 4: High Fidelity of Application’s Result Page

## 3.5 Development

Python served as the programming language, for developing and implementing the comparison tool. The choice of Python was driven by its collection of libraries for data processing and visualization along with its user nature. By leveraging the Flask web framework, it was able to enhance the tools capabilities and create an application that is lightweight. Flask provided functionalities for managing HTTP requests and generating HTML templates for user interaction. Various Python modules like Json, matplotlib, io and base64 were utilized to manipulate and display data effectively. Specifically, the functions "ctresources" and "generatebargraphs" analyzed JSON data to track resources count in each file and generated bar graphs using matplotlib based on usage counts. The index route encapsulates most of the applications features where uploaded documents are processed and resource counts are computed, graphs are generated, and HTML templates are applied to achieve desired outcomes.

The project repository is stored in the GitHub facilitating version control and easy collaboration. Adherence to coding standards was a focus throughout the development process to ensure consistency and readability of the codebase. Following PEP 8, the standard style guide for Python programming helped maintain a coding style across the project. Meaningful names, for variables, comments and documentation strings have been included to improve the code's readability and ease of maintenance.

## 3.6 Testing and Quality Assurance

A comprehensive testing strategy will be implemented to ensure the quality and reliability of the software tool. This strategy will involve two primary testing approaches. Each module and functionality within the codebase will undergo rigorous unit testing. Test cases will be designed to verify the intended behavior of individual features and functions under various input conditions. This granular testing approach helps to identify and address any issues early in the development process. Following code modifications and enhancements, regression testing will be conducted. A suite of comprehensive test cases will be executed to confirm that existing functionalities continue to operate as expected. This ensures that new features or bug fixes do not inadvertently introduce regressions or disrupt previously validated functionalities.

## 3.7 Iteration and Updates

Following each stage, changes are implemented only based on the results of testing. After development is complete, the software product is put through a thorough testing process to guarantee positive, functional, and usable results. Any faults or difficulties found are corrected in subsequent iterations based on the testing results. Every release that is created during the development process builds upon the previous one, with modifications and enhancements done only in response to testing results and changing needs.

## 3.8 Evaluation Metrics

Considered measures related to accuracy and usability to assess the usefulness of the instrument. Comparing rendering speed, accuracy measurements focus on the accuracy of comparisons. Usability metrics assess factors such as ease of use, intuitive navigation, user comprehension of the interface, and task completion time.

## 3.9 Ethical Considerations

Only certain users can access the Azure account holding the state file, ensuring that only authorized individuals may access confidential information. We reduce the possibility of unauthorized access or abuse of the data held in the Azure account by implementing stringent access restrictions.

To further improve data security and confidentiality, the repository containing the code for the comparison tool is kept up to date on a private GitHub account. This reduces the possibility of unwanted access or code manipulation by making sure the codebase is only available to authorized people.

## 3.10 Conclusion

To summarize, this approach consists of several phases, such as requirements analysis, development, and testing, all of which are in line with the study's objective of presenting a tool that illustrates infrastructure modifications after deployment. Combining experimental and applied research approaches, the study design takes into account the capabilities and limits of the tools. Implemented the development phases in accordance with the Waterfall software development model was ensured in an organized process. During the development process, requirements for choosing services for Infrastructure as Code (IaC) deployments were gathered via literature reviews. Version control guidelines were followed when creating the comparison tool with Flask and Python. Along with unit and regression testing, testing was centered on confirming functionality and stability with improvements mostly motivated by outcomes. Setting up security measures. Keeping a private repository is one way that ethical issues are addressed. In summary, this method successfully addressed the study's goal of examining how infrastructure configuration drifts in IaC deployments consistently by providing a framework for identifying and assessing.

# 4. Results and Discussions

In Figures 5 and 6, the project's real-time solution is displayed. A state file that is received upon deployment completion and a test file that contains required resource data are uploaded using the application's user interface, as seen in Figure 5.

A screenshot of a computer

Description automatically generated

Fig 5: Application’s Home Page

After the upload is complete, analysis is carried out and shown in graphical representations, as seen in fig 6. First, a text format is used to display the deployed resources' information. It provides information on the kinds of resources, how many there are, and which cloud environment each one was used in. Next, two bar graphs showing the differences between the actual and predicted outcomes are displayed. Each type's resource counts are displayed in both graphs. This application was developed with the implementation of the waterfall technique in an experimental and applied research methodology. Despite not having the flexibility of Agile methodologies, this approach supported systematic development and clear project requirements. The Flask web framework is used for application development, GitHub is used for version control and collaboration, Azure DevOps is used for continuous integration and delivery (CI/CD), and Python is used for tool implementation. Json, matplotlib, io, and base64 were among the other Python modules used for data processing, visualization, and user interaction.

A screenshot of a computer screen

Description automatically generated

Fig 6: Application’s Result Page

The application's performance metrics, such as average rendering speed and resource utilization are shown in Table 1. It states that the rendering speed is 12.89 seconds on average. Metrics for measuring resource consumption consist of percentages for CPU, memory, and disk intake. CPU usage is 1.4%, Memory consumed is 0.6% and the disk used is 0.3%.

|  |  |  |  |
| --- | --- | --- | --- |
| Average Rendering Speed | Resource Utilization | | |
| CPU | Memory | Disk |
| 12.89 secs | 1.4% | 0.6% | 0.3% |

Table 1: Application’s Performance Metrics

This study’s problem is addressed using Terraform state files, which allows us to compare the deployed infrastructure with the intended instances. As it can be seen in Figure 5, users submit a test file consisting of expected resource data and the state file, which is generated after deployment. After analyzing these data, the program shows the results in various forms, as it can be observed in Figure 6. The text output contains a brief description of the deployed resources. It includes the resource type and the count, the cloud environment with its specific type. Using this form, users now can easily see which resources are deployed and what environment they reside in. The bar graphs, which graphically depict the differences between actual and predicted resource configurations, provide more in-depth information. Through the comparison of resource counts on each axis, users are able to identify drift incidents and implement corrective actions. While desired configurations are not clearly presented in the text format, the issue statement is successfully addressed by the combined examination of all representations. It is easy for users to spot differences between planned and deployed states, which helps them keep their infrastructure consistent.

Based on Table 1's average rendering speed of 12.89 seconds, the application's performance is considered acceptable. This indicates that the technology can compare data fast and without generating significant disturbances [16]. Furthermore, modest resource utilization—CPU: 1.4%, Memory: 0.6%, and Disk: 0.3 indicate that the program functions effectively without requiring an excessive amount of processing power. These numbers demonstrate how helpful it is in identifying infrastructure drifts in real time. The most significant feature of this application is its user-friendly interface, which allows it to produce clear visualizations like text format and drift bar graphs. This helps in identifying differences in environments. However, there's always space for development. Despite the fact that both graphs show resource counts, differences aren't made clear by them. Clarity would be increased by using color coding, with green for matches and red for disparities. The program adheres to the principles of "user-centered design," which ensures an intuitive user interface [17]. Because all users will have on-demand access to this technology, it guarantees that its intended end users may utilize it freely and effortlessly. By sharply separating both of these functions from one another, the modularity principle may be used to separate data processing from data visualization [18]. Such a division could make maintenance easier in the future. A crucial lesson discovered throughout the project's development is the importance of carefully evaluating data presentation strategies that might improve graph clarity and prevent potential misunderstandings.

# 5. Conclusions and Recommendation

The goal of this project was to identify the drifts and achieved it by creating a JSON comparison tool that would make infrastructure administration easier. The application seems to be effective at finding configuration inconsistencies and displaying resource variations in understandable graph representations. This improved the efficiency of troubleshooting and easy detection. The tool can only process JSON outputs produced by Terraform scripts at this time. This restricts its ability to integrate with other popular IaC tools such as Chef and Puppet. In addition, because of resource limitations, testing was limited to a single cloud environment. This makes it difficult to determine if the tool will function accurately on different cloud systems and IaC formats.

Future work can help overcome these limitations by enhancing the tool’s capabilities. Among the tasks the first is increasing the support to generate graphs by analyzing data from other IaC tools. Additionally, comprehensive testing in different cloud configurations is crucial to ensuring the tool’s versatility. By identifying such drawbacks, this project lays the groundwork for the JSON comparison tool’s future work.

# 6. References

1. M. Chiari, M. De Pascalis and M. Pradella, "Static Analysis of Infrastructure as Code: a Survey," 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C), Honolulu, HI, USA, 2022, pp. 218-225, doi: 10.1109/ICSA-C54293.2022.00049.
2. M. Guerriero, M. Garriga, D. A. Tamburri and F. Palomba, "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 2019, pp. 580-589, doi: 10.1109/ICSME.2019.00092. keywords:
3. Matej Artač, Tadej Borovšak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. 2017. DevOps: introducing infrastructure-as-code. In Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17). IEEE Press, 497–498. https://doi.org/10.1109/ICSE-C.2017.162
4. H. Teppan, L. H. Flå and M. G. Jaatun, "A Survey on Infrastructure-as-Code Solutions for Cloud Development," 2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bangkok, Thailand, 2022, pp. 60-65, doi: 10.1109/CloudCom55334.2022.00019.
5. A. Mehdi and R. Walia, "Terraform: Streamlining Infrastructure Deployment and Management Through Infrastructure as Code," 2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS), Greater Noida, India, 2023, pp. 851-856, doi: 10.1109/ICCCIS60361.2023.10425616.
6. M. K. Bali and R. Walia, "Enhancing Efficiency Through Infrastructure Automation: An In-Depth Analysis of Infrastructure as Code (IaC) Tools," 2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS), Greater Noida, India, 2023, pp. 857-863, doi: 10.1109/ICCCIS60361.2023.10425162.
7. David Spielmann, Daniel Sokolowski, and Guido Salvaneschi. 2023. Extensible Testing for Infrastructure as Code. In Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2023). Association for Computing Machinery, New York, NY, USA, 58–60. https://doi.org/10.1145/3618305.3623607
8. Yiming Qiu, Patrick Tser Jern Kon, Jiarong Xing, Yibo Huang, Hongyi Liu, Xinyu Wang, Peng Huang, Mosharaf Chowdhury, and Ang Chen. 2023. Simplifying Cloud Management with Cloudless Computing. In Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets '23). Association for Computing Machinery, New York, NY, USA, 95–101. https://doi.org/10.1145/3626111.3628206
9. A. Ibrahim, A. H. Yousef and W. Medhat, "DevSecOps: A Security Model for Infrastructure as Code Over the Cloud," 2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC), Cairo, Egypt, 2022, pp. 284-288, doi: 10.1109/MIUCC55081.2022.9781709.
10. Nemania Borovits, Indika Kumara, Parvathy Krishnan, Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, and Willem-Jan van den Heuvel. 2020. DeepIaC: deep learning-based linguistic anti-pattern detection in IaC. In Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation (MaLTeSQuE 2020). Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3416505.3423564
11. Mohammed Mehedi Hasan, Farzana Ahamed Bhuiyan, and Akond Rahman. 2020. Testing practices for infrastructure as code. In Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing (LANGETI 2020). Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3416504.3424334
12. A. Rahman, E. Farhana, C. Parnin and L. Williams, "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts," 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, Korea (South), 2020, pp. 752-764, doi: 10.1145/3377811.3380409.
13. Daniel Sokolowski. 2022. Infrastructure as code for dynamic deployments. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 1775–1779. https://doi.org/10.1145/3540250.3558912
14. Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Automated testing of chef automation scripts. In Proceedings Demo &amp; Poster Track of ACM/IFIP/USENIX International Middleware Conference (MiddlewareDPT '13). Association for Computing Machinery, New York, NY, USA, Article 4, 1–2. https://doi.org/10.1145/2541614.2541632
15. Fairuz Iqbal Maulana, Vincentius Susanto, Putri Shilo, Julio Gunawan, Gusti Pangestu, and Dwi Riza Budi Raharja. 2021. Design and Development of Website Dr.Changkitchen Diet Catering Using SDLC Waterfall Model. In Proceedings of the 6th International Conference on Sustainable Information Engineering and Technology (SIET '21). Association for Computing Machinery, New York, NY, USA, 75–79. https://doi.org/10.1145/3479645.3479652
16. Jon Gunderson, Hadi Bargi Rangin, and Nicholas Hoyt. 2006. Functional web accessibility techniques and tools from the University of Illinois. In Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility (Assets '06). Association for Computing Machinery, New York, NY, USA, 269–270. https://doi.org/10.1145/1168987.1169049
17. Karen H. Jin. 2017. Teaching Responsive Web Design to Novice Learners. In Proceedings of the 18th Annual Conference on Information Technology Education (SIGITE '17). Association for Computing Machinery, New York, NY, USA, 115–120. https://doi.org/10.1145/3125659.3125684
18. Di Wang and Matthias Galster. 2018. Development processes and practices in a small but growing software industry: a practitioner survey in New Zealand. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18). Association for Computing Machinery, New York, NY, USA, Article 36, 1–10. https://doi.org/10.1145/3239235.3268926