National College of Ireland

IaC for Secure Serverless: A Comprehensive Approach to Deployment and Configuration Management

MSc Research Project
Programme Name

Hrishin Suresh
Student ID: 23159596

School of Computing
National College of Ireland

Supervisor: Shivani Jaswal

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Hrishin Suresh
Student ID:	23159596
Programme:	Msc in Cloud Computing **Year:** 2024
Module:	Msc Research Project
Supervisor:	Shivani Jaswal
Submission Due Date:	12/12/2024
Project Title:	IaC for Secure Serverless: A Comprehensive Approach to Deployment and Configuration Management
Word Count:	11306 **Page Count** 19

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Hrishin Suresh

Date: 12/12/24

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

IaC for Secure Serverless: A Comprehensive Approach to Deployment and Configuration Management

Hrishin Suresh
x23159596
Research in Computing
National College of Ireland

Abstract

Serverless computing has started to be adopted by more and more organizations in the last decade, but this widespread adoption has also introduced a slew of security challenges. Human error, a common cause of security vulnerabilities, can be significantly reduced by using Infrastructure as Code tools like Terraform to provision and manage infrastructure. This study evaluates the impact of using such tools to deploy a simple serverless application built on AWS Lambda. Potential threats will be identified using threat modeling frameworks and the Iac tools will be appraised on their ability to effectively minimize these threats. The experimental results show that security tools embedded in a pipeline identified 100% of intentional vulnerabilities, significantly outperforming manual inspection. The experiments also found that IaC tools reduced deployment time by over 80%. The research aims to establish a framework for secure IaC deployments in cloud-native environments, and also establish practical guidelines for developers to enhance security in their serverless applications.

***Keywords*— Infrastructure as Code, Serverless, Security, Automation, Pipelines**

1 Introduction

The start of this decade has seen a paradigm shift towards a cost effective cloud computing model where developers focus more on the application logic, while infrastructure management is abstracted. This has been enabled largely due to AWS Lambda, the leader in serverless computing frameworks which has enabled organizations to speed up their deployment cycles. Serverless technologies have started to be adopted by cloud platforms across the spectrum, showcasing the shift in how organizations are approaching scalability and operational efficiency of their services. According to the data published by Datadog(Datadog, 2023), over 70% of AWS and Google Cloud Platform users utilize at least one serverless solution in their infrastructure. The appeal of serverless computing is due to its cost saving abilities, lesser time to market and decreased reliance on server administrators. Despite these alluring benefits, serverless systems have their fair share of security challenges because of the limited control over the underlying infrastructure. The reliance on event driven models and shared resources has increased the attack surface, thereby exposing applications to threats like injection attacks and privilege elevations.

Traditional infrastructure management has had manually provisioned on-premises environments, and this faces many issues such as scalability, security and the speed of deployment. These limitations have driven people more and more towards the automated and dynamic approaches to management like using Infrastructure as Code (IaC). During infrastructure setup, the manual work like configuration of servers, network and storage are prone to human error. These errors can be of varying severity but ultimately they hinder the security and reliability of the infrastructure. IaC addresses these errors by enabling declarative, script based configuration, where the entire infrastructure is managed as a codebase. This allows automated and consistent deployments across a multitude of different environments, with minimal or no human intervention, thereby reducing human error. With IaC, any changes made can be versioned, tracked and a paper trail exists for auditing. This also enables easier rollback in case of any issues with updates.

Another consideration while building traditional systems is the planning and manual provisioning needed to handle any changes in demand. Scaling up resources involves physically adding or reconfiguring hardware in the case of on premises systems, or manually provisioning additional instances in the cloud, which can be time consuming. In case of systems with rapid changes in demand, the lack of instant scalability can lead to service disruptions. Serverless computing is one of the solutions to this issue, which allows automatic scaling in response to demand, with no manual inputs needed. AWS Lambda functions, which are a core part of this research, can dynamically allocate resources based on the incoming traffic. This ensures that any application can adjust to

fluctuating loads, improving the performance and reliability in any such situation.

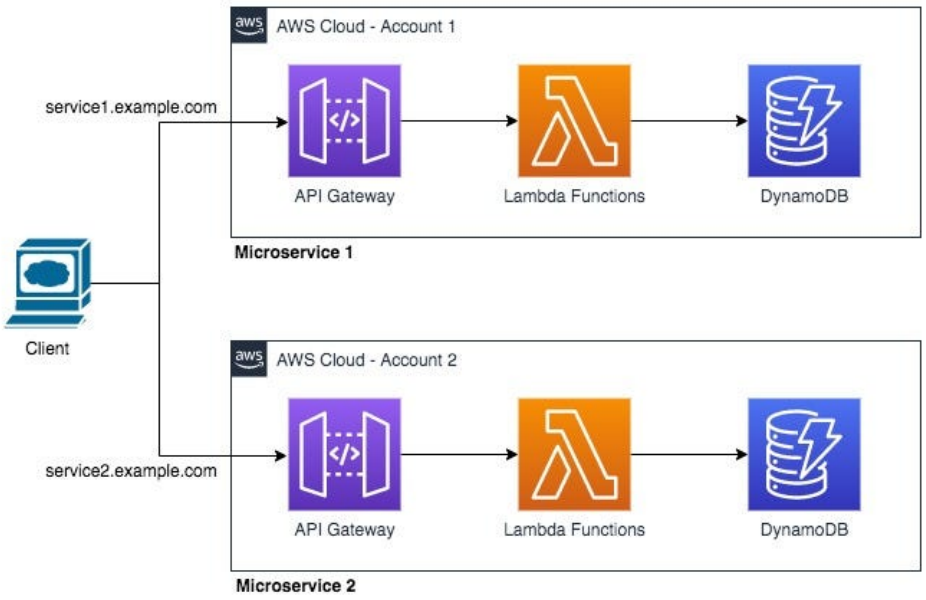

Figure 1: High level architecture of a simple serverless application(Waswani, 2020)

The figure 1 shows the high level architecture of a simple serverless application, such as the one used in this research. The client accesses a service, which calls an API. This API, in turn is connected to a Lambda function which performs operations on a database. In this way, a user is able to perform operations such as reading and writing data, without having a server as the middleman.

In most serverless architecture, the cloud provider abstracts most of the underlying infrastructure, making it impossible for developers to know what is going on in the backend. Though it has its advantages, this abstraction also means that developers have no access to the low level metrics and logs which are usually available in traditional environments. This limited visibility can make it harder for security teams to effectively monitor and detect anomalies or perform investigations into any incidents. IaC tools offer a partial solution to this issue by standardizing the infrastructure and configuration, ensuring consistent security settings are applied across all deployments. However, despite this, it is necessary to integrate additional monitoring tools to ensure robust security.

Serverless applications operate on an event driven model, i.e. functions are triggered by events like HTTP requests or APIs. This increases the attack surface, as malicious attackers can gain an entry point through each individual event trigger. IaC can be used to define standard security policies at each event trigger to help mitigate some of these risks. By automating these security controls and policies around every event handler, IaC will ensure that the security best practices are consistently followed across each event and function. The only drawback is that the user must be careful to avoid misconfigurations like authorizing a higher level of permission for a function that doesn't need it.

IaC scripts will often handle all configuration keys, API keys and environment variables. This must be properly secured so that there is no leak of sensitive data. Since these environments rely heavily on configuration, the testing must be thorough and comprehensive. Any gaps in the testing and validation of IaC scripts can lead to vulnerabilities. Addressing such vulnerabilities requires embedding security best practices directly into IaC. Practices like securing access keys, enforcing the principle of least privilege, and testing IaC scripts thoroughly before deployment will reduce the risk of misconfiguration with IaC.

This research will evaluate the impact of IaC tools such as Hashicorp Terraform on the security of serverless deployments. Using different threat modeling frameworks, the common security vulnerabilities will be identified and the ability of the IaC tools to mitigate these risks will be assessed. Threat modeling allows for the identification and categorization of security risks. Using threat modeling, the entire infrastructure can be mapped out and all the entry points for an attack can be identified. In a serverless system, this will include APIs, databases, cloud storage and event triggers.

Furthermore, this research will compare the different IaC tools to determine their effectiveness in enhancing the security of serverless deployments. Continuous integration and continuous deployment (CI/CD) pipelines will also be used to automate the testing and deployment process, further reducing the risk of human error and enforcing security practices at every stage. To implement and automate security checks effectively, this research uses Jenkins pipeline as the primary CI/CD tool. Jenkins provides a flexible, highly customizable platform with plugins for integrating security tools directly into the deployment pipeline. Using Jenkins, the security practices like static code analysis, dependency vulnerability scanning, compliance checks and so on can be embedded into the pipeline.

Several tools will be used in this study to create a secure serverless environment. AWS Lambda will be the core serverless function platform, working together with API Gateway. Data will be stored in DynamoDB and AWS S3, while CloudWatch will perform real time tracking of system activity, alerting for any unauthorized access attempts. Github will manage the version control for IaC scripts and pipeline configuration files. This will also ensure that all changes will be peer reviewed before it is added to the main branch. Docker will be used to create an image of the application before deployment. Security tools like SonarCloud and Aqua Trivy will be embedded into the pipeline. SonarCloud will ensure robust code quality and security and Trivy will specifically be used to scan the IaC configurations for vulnerabilities. CloudWatch will detect any runtime anomalies while Datadog will provide deep monitoring insights into AWS Lambda's performance and security. All these tools used together will provide a robust, automated and secure framework for deploying serverless applications. This enables the evaluation of these tools in a real world environment.

As more organizations begin their transition to a serverless architecture, the importance of security measures in these environments becomes more and more evident. A good understanding of IaC, combined with the use of CI/CD pipelines can enhance the overall security of cloud native applications. This research aims to provide insights into the security challenges posed by serverless deployments and offers guidelines to strengthen their overall security.

1.1 Research Question

What impact does leveraging infrastructure as code tools have on the security of serverless deployments and configurations?

Research question and objectives:

- Identify the common security vulnerabilities in serverless architecture
- Appraise the effectiveness of the different automation tools in detecting and mitigating the identified vulnerabilities.
- Propose any best practices for secure IaC implementations that will be found through the experiments.

1.2 Outline of the Report

This report has been organized into seven key sections to explore the research topic:

- Introduction - This section introduces the concept of serverless computing in todays computing ecosystems and its widespread adoption. The security challenges of serverless environments is explored and the potential of Infrastructure as Code is highlighted as a possible solution to the security issues in serverless deployments. The research question and objectives are also clearly defined for the study.
- Literature Review - This section examines existing research on serverless security challenges and the role of IaC in addressing these issues. Some of the literature that was examined focused on the vulnerabilities found in serverless systems, the best practices to be followed while writing IaC code and case studies about the advancements made in the field of security in serverless architecture.
- Research Methodology - This section outlines the approach taken to investigate the research problem. The STRIDE and OWASP framework are explained in the context of the research. Along with this, the tools used for testing and the experimental setup are explained in detail.
- Design - This is a short section which explains the architecture of the system.
- Implementation - This section details how the serverless application was deployed from scratch using Hashicorp Terrform. The configuration of the Jenkins pipeline is explained along with the integration of all the security tools. The challenges encountered during implementation are also discussed along with the steps taken to resolve them. The experimental setup is also discussed in this section with details about each experiment.
- Results - This section presents the results of the experiment as the findings of the study. A comparative analysis of the serverless application's security with and without IaC is done.

- Conclusion and Future Work - The report is concluded with a summary of the findings of the research and the implications of the findings in real world scenarios. Recommendations for best practices and areas for future research are explored.
- Bibliography - This section contains all the references used throughout this research.

2 Literature Review

2.1 Security Issues in Serverless

In their study, Marin, Perino, and Di Pietro (2022) explored the issues faced by serverless computing systems and drew comparisons with traditional computing methods. They found that the attack surface was broader in serverless, event driven architecture due to the frequent interactions needed between the functions and the cloud interface they are running on. This has the added downside of complicating the enforcement of security policies. The sheer number of event triggers also increases the potential points of entry for attackers. Furthermore, serverless systems rely on a shared pool of resources, which causes a higher risk for a data breach. This research was limited by the restrictions on accessing cloud providers' security protocols. The researchers were unable to evaluate the protocols effectiveness. Marin, Perino, and Di Pietro (2022) also noted that "warm" containers which are kept ready for faster response times, usually have weaker security defenses than "cold" containers.

In another study conducted by Bhatt, Sharma, and Bhadula (2024), a threat model was introduced to classify adversaries targeting serverless environments into two types namely internal and external. Internal threats involve maliciously designed functions inside the cloud environment that have access to sensitive data and can perform denial of service (DoS) attacks. External threats, on the other hand, typically exploit user provided inputs through APIs to run unauthorized commands or inject malicious code. The researchers identified several common attacks such as code injections, broken authentication, exposure of sensitive data, XML external entity (XXE) attacks and cross site scripting.

Drosos et al. (2024) also collaborates this research by categorizing the bugs they identified in IaC into internal and external. The researchers identified six primary symptoms, including external configuration failures, misconfigurations, and crashes, which together make up a significant portion of IaC issues. They observed that external configuration failures are the most prevalent issue, usually caused due to network errors or shell command failures. Misconfigurations are a threat which is just as bad, as they allow IaC scripts to execute successfully but yield incorrect infrastructure states. The researchers go a step further and explore root causes of these issues, identifying system interaction and state handling issues as the primary cause of these issues. They highlight the need for standardized mechanisms to detect and prevent such errors. This research gives the valuable insight that mismanagement of the underlying system state or unexpected interactions between modules leads to issues which cannot be easily diagnosed.

The study by Drosos et al. (2024) also addresses the limitations of current IaC testing methods, which rely on traditional software testing approaches such as unit and integration testing. The researchers argue that these methods fail to take into account the edge cases as they have not been adapted to the dynamic, multi-environment dependencies of IaC systems, To address these gaps, the authors recommend the development of enhanced testing frameworks tailored to IaC. These frameworks should be focused on working with different system states and configurations without issues. Some of these frameworks have been discussed further in other parts of this literature review.

The work done by Ntentos et al. (2024) addressed a critical gap in the understanding of security in IaC systems, focused on the design level security practices that are generally overlooked. With IaC tools like Ansible and Terraform, the absence of clear architectural guidelines for security leaves these systems vulnerable. The researchers examined how developers understand IaC scripts security practices by comparing traditional code inspection methods against architecture models and metrics. Through a controlled study with 94 participants, the authors found that the use of visual architectural models significantly improved the understanding of security in the system. Their results suggest that a "big picture" view may be critical in mitigating human errors linked to manual inspection.

The researchers also discussed the impact of architectural security gaps that might be missed during manual code inspection. Unlike implementation level security, design level issues encompass broader security challenges, such as managing interaction paths and securing privileged access. The findings emphasize the importance of a more structured approach to security in IaC, moving beyond basic code checks. The study suggests that using architectural models provide a clearer way to assess security in IaC setups. This approach could help developers better understand security needs in cloud environments by focusing on the broader system design, not just individual code security.

2.2 Infrastructure as Code

Infrastructure as Code promotes collaboration among DevOps teams by enabling the automation of management, monitoring and provisioning of cloud resources. This method boosts the operational efficiency while having the added advantage of minimizing human errors and shortening the deployment time of an application. Since the IaC tool controls the entire setup and deployment of the infrastructure, it is essential to follow security best practices when developing IaC scripts (Ketonen, 2024). Adhering to these security guidelines helps in identifying and preventing vulnerabilities and other configuration issues early in the development process. One notable challenge identified by this research is that IaC tools have a tendency of being cloud-specific, which leads to vendor lock-in. Such dependency on a single cloud provider can introduce security risks if the vendor specific configurations are not fully understood and implemented.

The DOML framework (Chiari et al., 2024) offers a solution to this issue by offering a multi-layered architecture. This DOML framework abstracts technical specifics inherent in current IaC tools. By simplifying both infrastructure and application layers, DOML lowers the barrier to entry for IaC tools, making the deployment process simpler and more accessible. Users of this DOML framework can generate IaC scripts across various cloud providers by using a single unified model. Despite these advantages, the researchers point out that implementing IaC through DOML has its own challenges, such as maintaining the consistency of code across all the different cloud environments.

Self-Provisioning Infrastructures(SPI) extend the serverless computing paradigm beyond simple functions, encompassing the entire application execution environment. By automating infrastructure configuration, SPIs reduce the risk of human error and vulnerabilities. In traditional serverless applications, developers hold the responsibility of the management and provisioning of infrastructure. This human involvement poses the risk of misconfigurations and other security vulnerabilities. SPIs mitigate this risk by automating configuration through self-provisioning mechanisms. SPIs will make sure that the network related settings, protocols for encryption and IAM policies are applied accurately and uniformly(Nastic, 2024). This study provides an overview of security policy implementation within the SPI framework, highlighting how SPIs do not need manual intervention to establish and maintain secure connections throughout the different components of the infrastructure like storage and API gateways.

SPIs enhance security by implementing granular access controls, role based access management, and continuous compliance monitoring. This ensures that functions operate with minimal privileges, reducing the chances of malicious attacks via the functions. Current serverless platfors like AWS Lambda and OpenFaaS offer basic self-provisioning features, but they rely on analytical or heuristic approaches (Raith, Nastic, and Dustdar, 2023), which are not sufficient in dynamic environments. To address this, the researchers state the need for an AI/ML based approach to improve self provisioning capabilities, allowing serverless functions to operate more efficiently and adapt to changing conditions and workloads.

A contrasting approach is presented by Kamath, Vignesh, Darshan, et al. (2023) who developed a comprehensive framework that integrates DevOps practices that uses different tools for automation and monitoring to optimize cloud infrastructure management. By using IaC and GitHub Actions for CI/CD, the framework improves efficiency, reduces manual errors and simplifies the deployment of resources. The core of this framework lies in IaC, which automates infrastructure provisioning for repeated use in future deployments. It seamlessly integrates with Github Actions to automate provisioning processes, allowing users to initiate resource requests through a frontend interface, which then triggers Github Actions workflows for resource deployment. These Github workflows execute tasks such as extracting configuration data, running Terraform scripts, and validating deployed resources. Furthermore, the proposed framework tracks and analyzes the resource usage and costs in the cloud with the help of Grafana.

2.3 Challenges in Implementing Secure Serverless

Serverless computing is an emerging field and many of the security challenges specific to this domain are poorly understood. Li, Leng, and Chen (2022), in their research, discuss the obstacles they faced in securing serverless computing environments. The researchers noted that most of these issues are caused because of the short lived nature of the instances and the separation of different parts of the application. The key problems found by the researchers include ensuring strong resource isolation and monitoring the functions continuously, despite their short lifespans. Additionally, implementing robust security measures and the protection of sensitive data during execution are significant hurdles. These issues are further amplified by the weak isolation in containers and the reliance on cloud providers to securely manage the clients sensitive data. For better resource isolation, tools such as Firecracker and microVMs were suggested to strengthen security boundaries. In terms of security controls, the researchers proposed techniques such as workflow sensitive authorization, but these methods are complex and

hard to implement.

A different approach to enhancing security in serverless systems is with the integration of proactive defense mechanisms (Shuai et al., 2024). The study done by these researchers critiques the traditional reactive security methods and introduces ATSSC. ATSSC is a platform designed to build attack tolerance using redundancy, diversity, and other other dynamic defenses. The key features of ATSSC include diverse replicas of functions, cross-validation for result accuracy and dynamic function refreshing. These techniques were implemented in the research by using Kubernetes and Knative, enhancing the systems resilience against potential attacks.

Nedeltcheva et al. (2023) conducted an in depth study on the challenges faced while designing and producing Infrastructure as Code scripts. The study highlighted the primary challenge as the inherently dynamic nature of IaC environments, which demands continuous security validation that extends beyond the traditional practices. To overcome this issue, they created a framework called PIACERE which integrates various security tools. A Model Checker evaluates the resilience of the infrastructure, while the IaC Scan Runner conducts multiple vulnerability assessments on the external libraries and other files utilized in the IaC files. Additionally, monitoring agents will constantly oversee the deployed infrastructure and can perform self healing actions when needed. Another significant challenge identified by the researchers was the management of sensitive information. PIACERE addresses this by storing the sensitive data in a secure place named the vault. During script execution, this data is called using environment variables and is erased immediately after use to ensure security and confidentiality.

Automating serverless deployments is the preferred choice for any modern cloud environment. Westman (2022), in his research, presents a detailed approach to automating a small scale cloud environment using AWS services. He leveraged AWS Lambda as the core component for managing server instances. This study highlights that serverless solutions enable infrastructure to be dynamic, automatically scaling resources in response to the demand. The automation framework used in this research uses a combination of AWS services such as SQS for task queuing, EventBridge for scheduling and Step Functions for workflow orchestration. By using these tools, the author creates an automated environment capable of initiating and teminating instances based on pre defined parameters. Westman used operating hours as the major parameter for function termination. This approach minimized the idle resource costs by ensuring that the server instances would only operate in the designated hours, which is a significant improvement when considering that all this happens seamlessly without any interaction. The setup proves how automation can streamline the infrastructure based on the usage requirements and can be scheduled to remove the need for any manual interactions.

The research also provides insights into the limitations of serverless tools. Westman (2022) noted that tools like CloudFormation offered declarative infrastructure management, but AWS Lambda's procedular nature was better suited to the project's dynamic requirements. Tools like CloudFormation require careful error handling, but the flexibility they offer in dynamic environments is unmatched. Many different tools were used by the researcher to showcase a mixed approach to serverless automation, using both procedural and declarative tools wherever appropriate. This mixed approach ultimately allowed the deployment of a robust and adaptable cloud environment. However, this research was done in a small-scale environment. Although valuable points were raised, further testing could have been done to analyze the cost effectiveness and performance of such a system on a large scale environment.

2.4 Security Testing in IaC

The integration of automated security tests into Infrastructure as Code(IaC) practices is crucial for maintaining security throughout the software development lifecycle. Sapkota (2023) highlights the value of integrating security tools directly within CI/CD pipelines, making sure that security checks are conducted during every phase of the development. This study evaluates different security automation tools including Snyk, Synopsus Code Sight, GitHub Advanced Security, GitGuardian, OWASP ZAP, Nessus, W3AF, Burp Suite and Nikto. The metrics used for their analysis were tool performance and effectiveness in identifying security vulnerabilities.

The study also discusses the challenges faced while implementing security testing tools within IaC workflows. Due to the lack of standardization of IaC code, scripts are written differently across various tools and cloud providers. This variability, along with the intricacy of IaC scripts creates barriers to embedding security tools into IaC workflows. However, the researchers argue that the benefits of using these tools– such as enhanced security, early detection of issues, increased efficiency and regulatory compliance—far outweigh these challenges.

The researchers also explored numerous IaC security testing tools used for analyzing code and identifying issues prior to deployment. Tools such as Terraform, Ansible, Chef and Pulumi were reviewed for their ease of use and wide array of features which facilitated secure infrastructure management. Out of all the tools reviewed, Terraform stood out due to its extensive support, adherence to security best practices, capabilities for integration

and the detailed documentation that is available on their website. The research provides invaluable insights into secure coding practices, continuous security testing and the strategic use of automation tools for security.

While automated security tools address specific vulnerabilities, comprehensive testing methods are needed to verify the consistency and compliance of IaC scripts across various deployment environments. This necessity has caused researchers to explore advanced testing frameworks such as the Automated Configuration Testing(ACT). Sokolowski, Spielmann, and Salvaneschi (2024) finds that the current testing methodologies for IaC programs are limited in their practicality and efficiency. Despite the effectiveness of the traditional unit testing approaches, the high effort needed is a deterrent. Integration tests are less demanding to code but have high time and resource costs, making them infeasible for high velocity IaC deployments. The researchers identified this gap and proposed Automated Configuration Testing, a unit testing framework designed to streamline IaC testing by automatically generating mocks and validating configurations with the help of oracles. ACT was implemented in the ProtoTI tool for Pulumi in order to enable rapid, automated testing without any coding requirements.

In their evaluation, the researchers found that ACT consistently demonstrated high efficiency in identifying bugs across various configurations. A large sample of IaC programs was used for their testing. Sokolowski, Spielmann, and Salvaneschi (2024) validated ACT's reliability by drawing comparisons to conventional IaC testing methods and confirming its rapid detection of bugs, especially in edge cases. Furthermore, the pluggable architecture of ACT allows the integration of third party tools for further customization, in order to adapt to evolving IaC environments. This research provides a foundational framework for improving IaC reliability for cloud native applications.

2.5 Automating Serverless Deployments

With the rise of serverless computing, automation of configuration processes has become essential for optimizing the cost and performance of these environments. An innovative approach was taken by Moghimi (2024) for automating serverless configurations. The research addressed several critical developer challenges like resource allocation and security settings. The study uses a tool called Parrotfish to automate the process of resource resizing for serverless applications by using regression models to determine the optimal configuration of resources. Parrotfish allows developers to achieve cost effective deployments without affecting the performance, which in turn leads to easier management of serverless environments.

Another challenge to serverless automation is the security configuration, which can lead to vulnerabilities and attacks. Moghimi (2024) addresses this gap with Growlithe, a tool which automates the generation of security policies to ensure that each serverless function has only minimal permissions needed for their execution. This automation improves security by removing human error from the configuration process and providing fine grained, automated tracking for security policies.

With the help of both these tools together, the researchers envision the potential of automation to deliver a truly serverless experience, where the developers can focus on the working of the application itself rather than the complex infrastructure running behind the scenes. This research highlights how automation tools save the time and effort spent on managing resource and security configurations, in turn making serverless deployments more accessible.

2.6 IaC Best Practices

Kumara et al. (2021) provides a detailed review of best and bad practices in Infrastructure as Code, focusing on essential guidelines for writing effective IaC scripts. The authors stress the importance of writing code that is easy to understand for other developers. Emphasis was given to clear naming conventions, consistent coding styles and explicit parameter declarations. The authors found that enhancing human readability simplifies collaboration, debugging and long term maintenance.

Another key practice that has been discussed is modularization. By organizing the IaC code into modules, infrastructure managers can minimize code modifications required during future deployments. The authors also emphasize the value of a versioning control system to track code changes, as well as the need to separate configuration data from the code itself. The authors recommended the use of a dedicated datasource for the configuration files. The study presents a structured breakdown of best practices, organized into 10 primary categories and a total of 33 subcategories.

The paper also identifies several "bad practices" in IaC, grouped into four main categories. Automation is highlighted as a core principle, with the authors advising that all manual steps should be automated within IaC

source code to adhere to the IaC standards. Although often overlooked, naming conventions are noted as critical, as inconsistent names can lead to resource confusion and management issues. Unnecessary code complexity is also discouraged. Simpler code enhances readability and reduces the potential for errors. Avoiding these poor practices is crucial for improving security and efficiency in IaC ecosystems.

3 Research Methodology

3.1 Threat Modeling

To perform threat modeling on this framework, first we must map out the data flow and dependencies of each layer using a data flow diagram. The data flow diagram of the serverless application is shown in figure 2. As found in the research done by Ntentos et al. (2024), a big picture of the architecture of the system is crucial in understanding security needs.

The users of the serverless application get the frontend of the website served to them through the S3 bucket. On interacting with the front end via either a GET or POST request, the API gateway is triggered. API gateway then resolves this request and routes it to either of the two Lambda functions. The insertStudent function is triggered when it is a POST request and the getStudent function is triggered when it is a GET request. Both these functions can then access the DynamoDB table to perform the POST and GET functions to the table respectively. This entire system will be launched through an Infrastructure as code script running in a Jenkins pipeline.

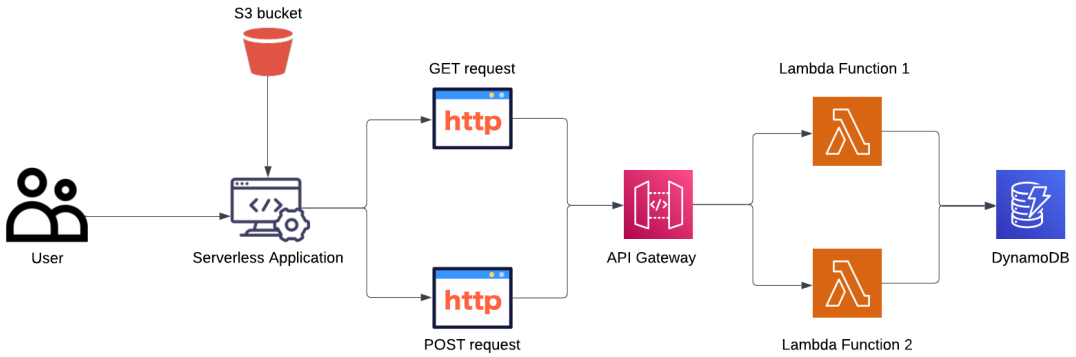

Figure 2: Data Flow Diagram (original illustration)

For this research, we will consider the STRIDE threat modeling framework. The system architecture has already been defined in the previous paragraph. STRIDE stands for Spoofing, Tampering, Repudiation, Information Disclosure , Denial of Service and Elevation of Privilege. Each of these security threats will be explored in more detail below.

- Spoofing - Spoofing in a serverless system could be either a malicious user impersonating as the AWS user or Jenkins pipeline administrator. This risk can be mitigated by using strong authentication like secure IAM policies and API keys in AWS. For the Jenkins pipeline, multifactor authentication can be implemented to make sure that the admin access cannot be spoofed. Datadog has been used to monitor access patterns and detect any anomalies that may occur.

- Tampering - Tampering in the context of this research could refer to alteration of the terraform files or the jenkinsfile which controls the entire Jenkins pipeline. To keep a paper trail of tampering and also to help prevent it, the code is stored in a repository on Github. The pipeline uses only this main repository where any and all changes are tracked with the git version control system. Github also has authentication tokens and multi factor authentication to further prevent any unauthoized parties from accessing the code. The terraform state files are files which are generated once the infrastructure is launched. This state file contains all the details of the current deployment, and any changes made to the deployment will reflect in the state file. This file can be encrypted or stored in an encrypted S3 bucket to prevent unauthorized parties from accessing this sensitive information.

- Repudiation - Repudiation is the lack of evidence to trace an action. In this research it can refer to the changes made to the pipeline or in the infrastructure. The changes to the pipeline are already secured and have a paper trial through the use of Github. However, the infrastructure can be manually altered through the AWS console without ever seeing or modifying the Terraform code. To account for this, CloudTrail logs can be enabled. Cloudtrail is an AWS service which records actions taken by every user, role and service as an event that can be logged (AWS, 2024). This helps with auditing and compliance of the AWS account. This service will ensure that there is repudiation in the AWS account.

- Information Disclosure - This refers to exposing sensitive information like the terraform state files or the environment variables used throughout the code. To account for this, the terraform state file has been encrypted. AWS Secrets manager can be used to store all the environment variables securely in accordance with secure coding practices.

- Denial of Service - The service availability of a serverless system can be disrupted by overloading API Gateway or repeatedly deploying misconfigurations through the pipeline. This can be protected against by rate limiting API Gateway in the settings. Checks can also be implemented in the Terraform Plan stage to stop Terraform from applying the configurations if they are unsafe. Datadog has also been used to monitor the lambda services for any anomalies.

- Elevation of Privilege - In the context of this research, elevation of privilege will refer to over permissive IAM roles for the Lambda functions or in the Jenkins pipeline. Principle of Least Privilege [POLP] has been followed throughout the research project for every IAM role. All services have the minimum required permissions for their execution. This ensures that there is no service which can perform any unauthorized or unwanted actions.

The STRIDE framework is a good general guideline for ensuring robust security in a system, but to go more in depth for specific security vulnerabilities, this research has incorporated the OWASP Top 10 security risks. The OWASP Top 10 OWASP, 2024 is a report of the top 10 common security vulnerabilities that plague web applications and APIs. The combination of STRIDE along with OWASP will ensure that the general security vulnerabilities are addressed along with the specific vulnerabilities that occur in practical situations which can compromise the systems reliability. Since some of the vulnerabilities on OWASP are the same of similar to the STRIDE discussion done above, they will only be explained briefly or skipped entirely.

- Broken Access Control - Broken Access control is the unauthorized access to sensitive data or resources. This was found to be the most common security vulnerability by OWASP. The methods to counter this have been discussed extensively in the previous section.

- Cryptographic Failures - Cryptographic failures occur when sensitive data has weak encryption or is not encrypted at all. This includes terraform state files, environment variables and so on. This is accounted for by storing the state files in an encrypted S3 bucket. This way, the onus of the encryption is on the cloud service provider i.e. AWS. The Sonarcloud analysis stage in the pipeline has been used to identify any missing encryption in the configuration files. If any environment variables are exposed, sonarcloud will pick those up in the analysis.

- Injection - Injection occurs when a malicious input string causes unintended behaviour. In the context of this research, this can occur when API gateway forwards unvalidated data to the AWS Lambda functions. To address this, input validation and sanitization is used. Aqua Trivy has been implemented in the pipeline to scan for any vulnerabilities or dependencies that may enable injection attacks.

- Insecure Design - Insecure design refers to any security flaws or design flaws. This can refer to any policies that have been misconfigured or infrastructure that has been configured without security policies. Aqua Trivy has been configured to detect any such configuration issues in the Infrastructure. If any such issues are detected, the pipeline will fail.

- Security Misconfiguration - Security misconfigurations is similar to the previous point of insecure design. It can refer to misconfigured Terraform files, publicly accessible S3 buckets or API Gateway functions. Aqua Trivy and Sonarcloud have been used to validate Terraform and Jenkins configurations via the Jenkinsfile.

- Vulnerable and Outdated Components - The effect of this vulnerability can only be seen via the Lambda functions using vulnerable python packages. To prevent this, Trivy can scan and update the dependencies

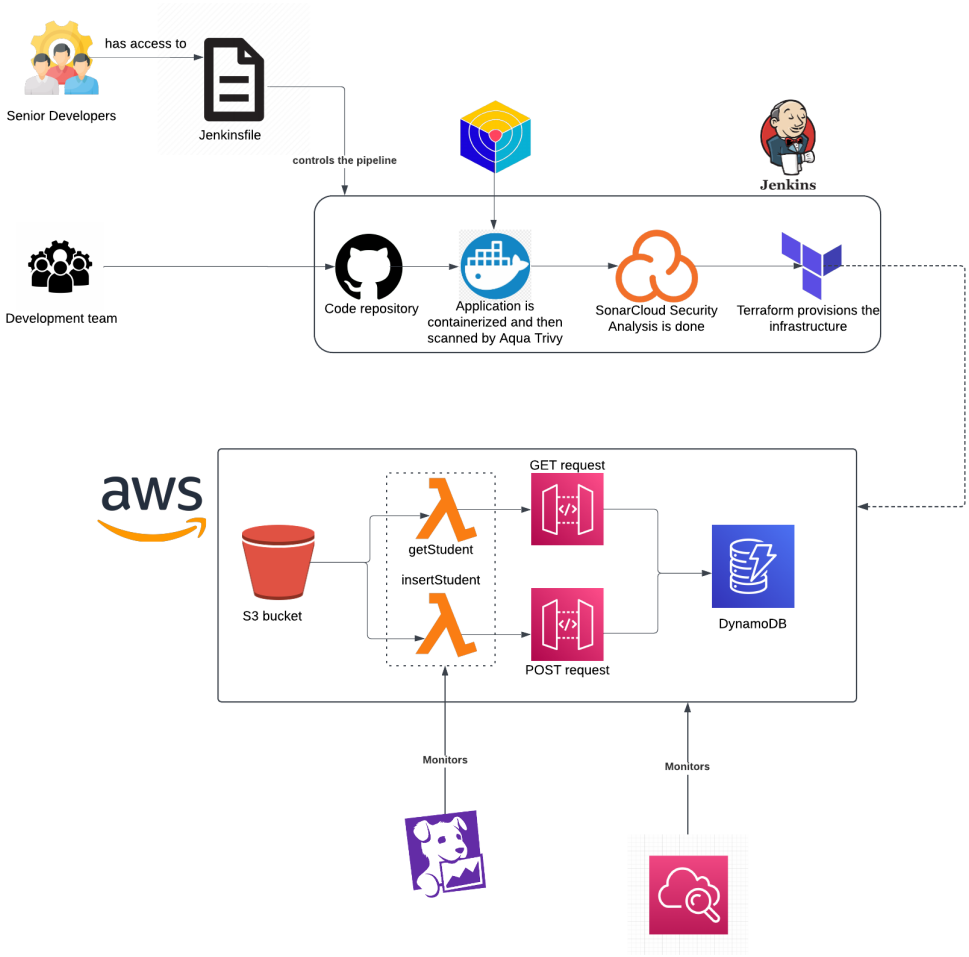

Figure 3: Architectural Diagram

in Lambda. CloudWatch has also been set up for real time monitoring to detect if any vulnerable packages are getting executed.

- Security Logging and Monitoring Failures - One possible effect of this is that changes to infrastructure will not be logged in AWS. This has already been discussed in the previous section and AWS Cloudtrail can be used to create logs of any such activities.

- Server-Side Request Forgery - This refers to exploiting server side services to access internal resources. Since it is a serverless deployment, the only threat from this is the Lambda functions being exposed to vulnerabilities in the HTTP requests. To prevent it, the network traffic can be monitored in Datadog to detect any abnormalities.

With STRIDE and OWASP top 10, it can be said that most of the security vulnerabilities are accounted for and prevented. In order to test the effectiveness and evaluate this research, these security threats will be simulated and the effectiveness of IaC tools can be measured in mitigating these threats.

4 Design Specification

The high level architectural diagram for the framework has been showcased in fig 3. The jenkinsfile is the most important file since it controls the entire pipeline. If the file is not secured, anyone can make changes to the pipeline and compromise the security of the entire serverless deployment. Hence in this framework, only the senior developers will be able to access the Jenkinsfile. The entire development team will have access to the rest of the code repository. The code repository has been hosted on GitHub. When the pipeline is executed, the code

is checked out from GitHub and containerized into an image using Docker. Aqua Trivy then scans this docker image along with all the IaC files inside it and provides a full vulnerability report which can be accessed via the Jenkins dashboard. If any vulnerabilities are detected, the pipeline will fail. After the trivy scan, the pipeline moves on to the Sonarcloud analysis stage which performs a sonarqube quality gate assessment. If the quality gate has passed and the deployment is approved by the user, Terraform will provision all the infrastructure to the linked AWS account. The application uses a html file hosted on a secure S3 bucket for its frontend, along with 2 lambda functions - getStudent and insertStudent as the basic functions for the get and post operations to the database. The database that is used is DynamoDB and two separate API gateway methods are set up for GET and POST requests. Datadog has been set up to monitor the Lambda functions for any suspicious behavior i.e. to detect any DDoS attacks via the network usage or any other abnormal instances. In addition to this, CloudWatch has been set up for real time monitoring of all the services along with alarms to notify the user in case of any abnormalities. The working of the pipeline has been explained in depth in the Workflow section of this report.

5 Implementation

The research experiments have been set up to evaluate the impact of IaC tools on the security of serverless deployments. The cloud platform used for this demonstration is Amazon Web Services, which is the cloud provider with the biggest market share with over 31% of the global market share (Richter, 2024). The serverless components used for the deployment are AWS Lambda, API Gateway with 3 routes - GET, POST and OPTIONS and an S3 bucket. A private VPC has been set up, with specific IAM roles created for DynamoDB and the Lambda functions. The monitoring tools used are CloudTrail and Cloudwatch for monitoring AWS and any changes in the environment. Datadog has been set up to monitor the Lambda function along with the network usage by AWS.

5.1 Tools Used

This section will outline all the tools used throughout this research undertaking.

- Terraform - Terraform is the core component of this research. It is the Infrastructure as Code tool used for the purposes of this project. This tool was developed by Hashicorp for the provisioning and deployment of infrastructure using code.

- AWS Lambda and API Gateway - Lambda is the pioneer of function as a service tools in the market today. These two services in conjunction will form the crux of the serverless application used in this research.

- AWS S3 - An S3 bucket has been used to store the data required for hosting the application. This encrypted S3 bucket also contains other sensitive files like the jenkinsfile and the terraform state file. S3 was selected due to its integration with the rest of the AWS suite and its encryption capabilities.

- AWS CloudWatch and CloudTrail - CloudWatch provides a comprehensive view of the logs which are generated by the different AWS services being used. Alarms can be set up in CloudWatch to trigger when any unauthorized actions are detected. CloudTrail is a similar service which records any actions taken by a user in the AWS console. This service proves very useful for auditing and compliance reasons.

- GitHub - GitHub is the versioning control system that has been used for the purpose of this research. All the configuration files, IaC files and Jenkinsfile have been pushed to a github code repository. This repository creates a paper trail of the modifications done to the code as well as to the jenkinsfile which controls the entire pipeline. In a real world setting, any changes to the codebase will have to be peer reviewed for added security before deploying it to production.

- Jenkins - Jenkins is the open source automation tool that has been used to set up a pipeline in this research undertaking. Since it is open source, it is secure and has seamless integration with all the security scanning tools used in this research via downloadable plugins. Jenkins provides an easy configuration capability for CICD using a file called a Jenkinsfile which is used for defining the pipeline. The entire pipeline can be controlled using this Jenkinsfile and any changes can be made in the file itself. This file needs to be secured and encrypted. If a malicious user gets access to it, they can mess up the production server.

- Datadog - Datadog is the monitoring tool which has been integrated specifically to monitor the Lambda functions for any suspicious activity. Datadog can generate enhanced metrics for Lambda providing detailed data on its invocations, duration and errors in real time. All these metrics are visualized in an easily

understandable dashboard, which provides insights into the performance and security of the AWS service.

- SonarCloud - SonarCloud is the cloud based offering of SonarQube, which helps to improve the code quality of a repository by detecting code smells and vulnerabilities. This tool provides reports which ensure that good coding practices are followed early on, removing the need to refactor code later.

- Aqua Trivy - Trivy is a comprehensive vulnerability scanner which can scan all code artifacts, Docker images and IaC scripts. Trivy is used in this research due to its ease of integration into the Jenkins pipeline and its ability to detect issues with IaC scripts as well as security vulnerabilities.

- Docker - Docker was used to containerize the application and its dependencies, ensuring that there is consistency between testing and production environments. It also added to the security of the application due to Trivy being able to perform comprehensive vulnerability checks on Docker images.

5.2 Workflow

This subsection will outline the approach taken for this research, explaining how the IaC scripts and Jenkins pipeline deploy the serverless application securely.

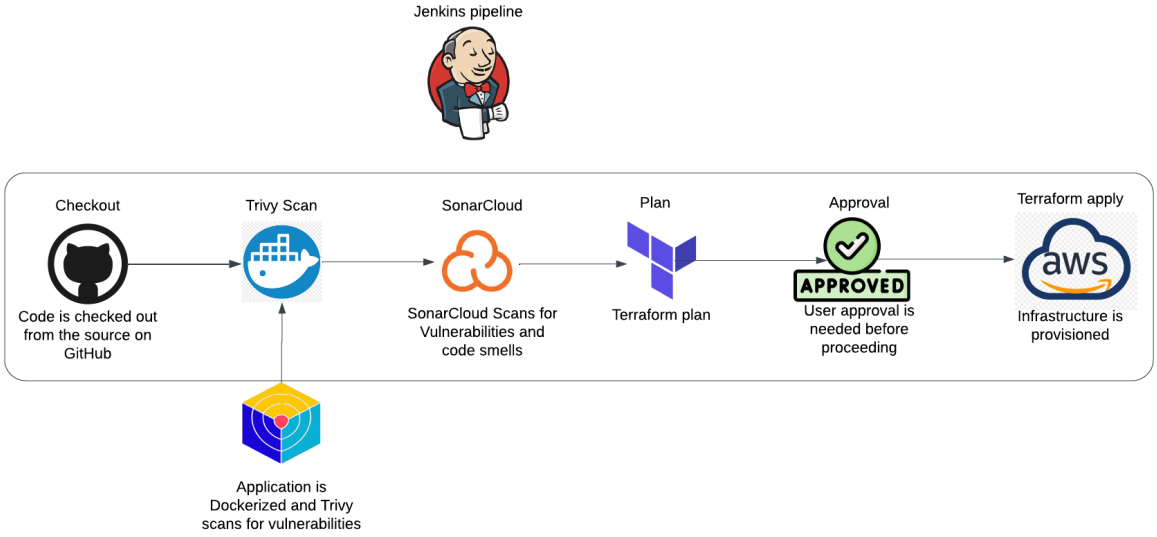

Figure 4: Jenkins pipeline stages (original illustration)

The IaC scripts were written in VSCode using Hashicorp Configuration Language[HCL] to write Terraform Scripts. The Terraform version used was 4.0 which had major improvements to S3 bucket configurations for AWS. The AWS account first needs to be linked using the aws cli. All the resources for AWS were provisioned in the eu-west-1 region for minimal latency during testing. The IaC script deploys a DynamoDB table, S3 bucket, 2 Lambda functions and an API Gateway with 3 routes - GET and POST, as well as a mock OPTIONS route to prevent CORS errors. Along with these, The IaC script is also responsible for assigning policies and permissions to all these services. The index.html and scripts.js file is uploaded to the s3 bucket automatically using the IaC script itself. It is important to focus on security at every stage, so environment variables have been used to protect sensitive data, as well as using secure coding practices and strong security policies for the infrastructure.

Once the IaC code is written, the code for the serverless application is needed before it can be provisioned. A simple index.html and scripts.js file serves as a frontend for the application. Two python scripts namely the getStudent and insertStudent serves as the backend for the serverless lambda functions. The application is a simple serverless application which inserts and retrieves student data from a noSQL database. The two python scripts are zipped using 7zip compression software so that Terraform can use these files. This entire code is pushed to the Github repository for further use.

This completes the basic serverless application deployment using IaC. Now for security, a Jenkins pipeline is implemented. A Jenkins server with the relevant plugins is hosted on the local system to run the pipeline. The

Jenkinsfile defines the different stages of the pipeline. The first stage is the checkout stage which is linked to the GitHub repository where the code is residing. This stage remains the same for most Jenkins pipelines. In the second stage, Aqua Trivy is incorporated to scan the project. Since Trivy works well with containers, the entire project can be containerized into a docker image, and then Trivy scans the entire image with special attention to the IaC script. Jenkins has a plugin for Docker which allows building and pushing Docker images to a private registry. Once this image is scanned, Trivy generates a report and saves it into a .txt file which can be accessed through Jenkins when the pipeline is run. The pipeline is configured to fail if the Trivy scan finds any HIGH or CRITICAL errors in its scan. The next stage of the pipeline implements SonarCloud. SonarCloud is linked to the repository using its web interface. However it is implemented in the pipeline so that the SonarCloud scan is run everytime the pipeline runs. If the quality gate of SonarCloud fails i.e. if there are any errors, issues or vulnerabilities with any of the code in the repository, the Pipeline will fail.

When both these security checks are complete, the pipeline moves to the plan stage where the terraform file is initialized. terraform init and terraform plan commands are run to see if the configuration of the infrastructure is correct. If everything works as expected and there are no errors or dependencies, the pipeline moves to the approval stage. In this stage, the user's approval is required to deploy the IaC script. The output of terraform plan is shown to the user and the user can review this and approve or deny the plan. For ease of testing, there is also an auto approve setting that can be checked in the Jenkins pipeline before it is run. Once approved, the pipeline moves to its final stage, the apply stage. In this stage, the terraform apply command is run which provisions the infrastructure in the linked AWS account. This can be confirmed by checking the AWS management console of the account and checking to see the infrastructure. The url of the S3 bucket is also provided as an output of the pipeline for easier access.

One of the main errors faced during this implementation was that of CORS- Cross Origin Resource Sharing. Even though CORS is enabled via code in the main.tf file, the CORS on the API gateway was stuck on disabled. The issue was found to be due to the routing of the API gateway. The CORS issue was occurring because the method responses for the api gateway GET and POST methods were not being configured even though they were declared explicitly in Terraform. A workaround for this was to manually configure method responses with CORS headers from the AWS console after executing the IaC script. There was also multiple vulnerabilities detected by Trivy due to the Debian version used by Docker. Due to this, the python version had to be manually updated from version 3.10 to the latest supported by docker, however even the latest version had vulnerabilities detected by Trivy causing the pipeline to fail, hence these CVEs were added to a .trivyignore file for the purposes of the research.

5.3 Experimental Setup

In order to evaluate the impact that Infrastructure as Code tools have on serverless deployments, a structured approach was taken which involved a series of experiments. The experiments that were conducted focused on metrics like deployment efficiency, error rates and the security vulnerabilities that are detected by the pipeline. These experiments will provide insights into the benefits of using IaC in a secure serverless architecture, as compared to manually provisioning the infrastructure.

5.3.1 Time Taken for Secure Iac vs. Manual Deployment

The objective of this experiment is to directly compare the time taken to deploy a serverless application manually versus using an automated CI/CD pipeline integrated with security tools.

For manual deployment, first an encrypted S3 bucket was created on an AWS account to store all the files required. Then the bucket settings must be configured, including generating a policy for it using the bucket policy generator, along with CORS settings and allowing public access to the bucket. Once this is done, a DynamoDB table is created with the required primary key along with a test input. Next, two Lambda functions have to be set up. The programming language selected for this research was Python, so Python code for both the functions were copy pasted into each Lambda function. Next, the required IAM roles and permissions for Lambda to access the DynamoDB table have to be set up. API Gateway can then be set up on the console with GET and POST requests, linking them to the specific Lambda functions. CORS is then enabled on the APIs and the API is deployed and the endpoint is generated. This endpoint is then required in the scripts.js file which will be uploaded to the S3 bucket along with the index.html file required for the frontend of the application. This is how the application is manually deployed.

Once deployed, there is some testing required to make sure that everything works as expected. The easiest way is to test the Lambda functions using the test feature in the dashboard, and then the API gateway endpoints, and finally testing the application itself. In case of any issues, they must be manually debugged via the interface. In order to measure the time taken for this approach, a timer was started on initiating the creation of the first

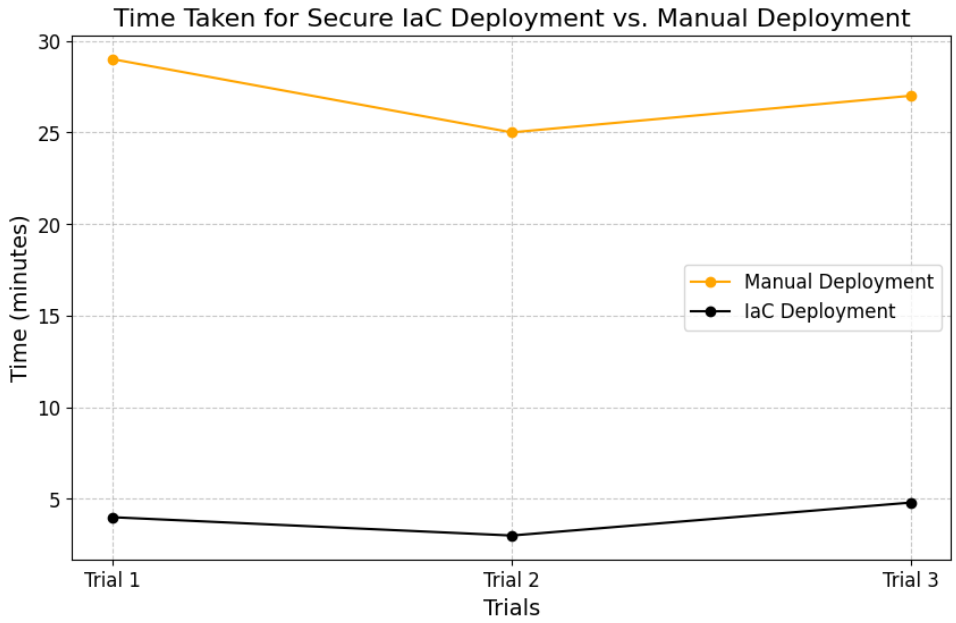

Figure 5: Time taken for IaC vs Manual Deployment [Exp 1]

service, and it was stopped once every setting was tested and the application was confirmed to be working as expected. This deployment was carried out 3 times to account for variability and internet speeds and the average is shown in the graph.

The automated deployment is much simpler but requires an initial set up of the pipeline and IaC script. Firstly, a Terraform script must be written to configure the required infrastructure which includes an S3 bucket, Lambda functions, API gateway and DynamoDB along with all the necessary roles and permissions needed for each of these services to work in the context of a serverless application. Since this code is going to be uploaded to GitHub, it is important to use environment variables for data like access keys and endpoints. Once the script is ready, a Jenkins pipeline can be configured with a checkout stage for pulling the script from the GitHub repository, a stage for creating a docker image and scanning it with Aqua Trivy, another security scan stage for SonarQube, followed by a plan stage to preview the infrastructure and then an approval stage to proceed with provisioning the infrastructure. In this approach, the security tools like Trivy and SonarCloud detect any issues and misconfigurations in the code, which has the added benefit of making debugging easier.

The time taken for this deployment on the first attempt is much higher due to the prior setup required, but for the experiments, the time has been measured from the start of execution of the pipeline till the infrastructure is tested on the AWS console and the application is found to work properly. This deployment was carried out 3 times to account for variability.

The results of this experiment are shown in fig 5. We can infer from the graph that using IaC tools significantly reduces the time taken for deployment. In the experiment, IaC was able to deploy and run the infrastructure over 5 times faster than using manual deployment methods. Even destroying the resources was found to be a lot easier and faster by using the "terraform destroy" command as compared to manually going to each service and deleting them. The results for each experiment will be analyzed further in the results section.

5.3.2 Configuration Error Detection in Manual vs Automated Deployments

This experiment is conducted to measure and compare how many configuration related errors can be detected and mitigated through the use of automated deployments with IaC tools. This will be compared against the same errors having to be detected in a manual deployment.

For the manual deployment, the same serverless setup was used as in experiment 1, however, some deliberate misconfigurations were introduced. Higher privilege was given to the Lambda functions in the form of AdministratorAccess role and the API Gateway routes have been misconfigured and are not linked to anything. The application is not able to interface with the database due to this. Since this is a manual deployment, the error is only known if the user knows where to look. A step by step approach had to be taken to test this by first

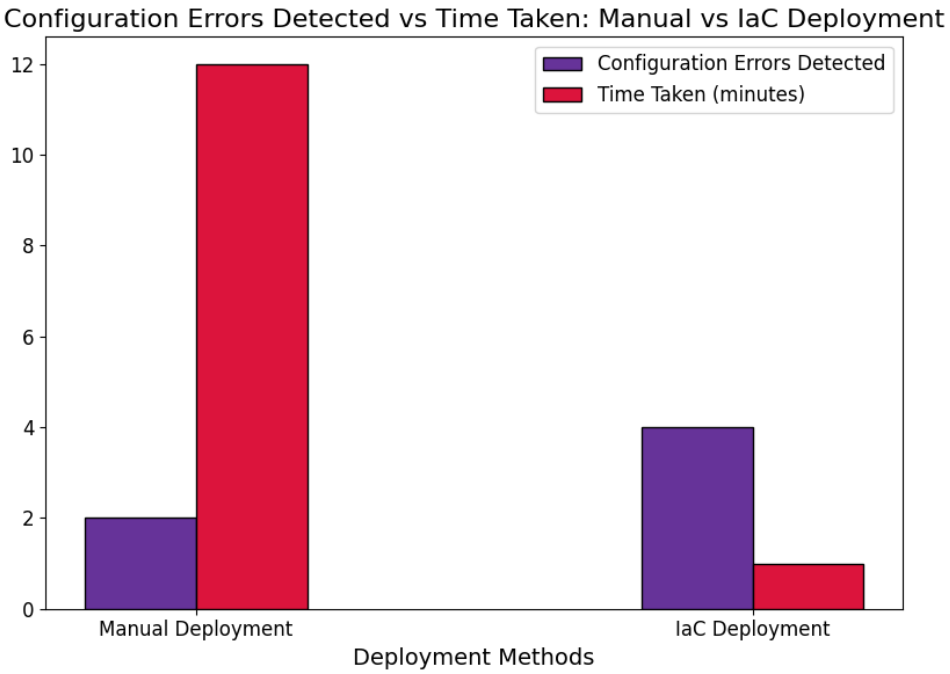

Figure 6: Configuration Error Detection in Manual vs Automated Deployments[Exp 2]

checking the logic of the Lambda functions and then once that is confirmed to be working correctly, the API Gateway routes are examined and the error is found. Since this is a small application, the error was found on the second step, however in large deployments, a manual approach will be very time consuming and hard to find. The over privileged lambda function is not found out as there is no situation where it stands out in the deployment, therefore the administrator will not be aware that it is a misconfiguration.

In case of the automated deployment, the necessary changes are made to the IaC script and the pipeline is executed. Terraform itself picked up the misconfiguration, alerting the user that the API gateway was incorrectly configured, followed by Aqua Trivy showing the same alert along with a warning about using AdministratorAccess only when necessary. An unused IAM policy was also picked up Terraform and highlighted as an error. The publicly exposed API endpoint was also initially shown as a warning in Sonarcloud during setup.

In the experiment, the IaC tools were able to find more configuration errors than the manual deployment, in a much lesser time. The results of this experiment show that IaC tools can significantly reduce configuration errors in the final product, because the misconfigurations are validated and ironed out during the provisioning phase itself, so they never make it to the production environment. It is also found that manual process rely heavily on the user's experience and expertise in the field. Common issues can be found by most people but only users with good attention to detail and experience will be able to find out all the misconfigurations in a deployment without the help of any external tools.

5.3.3 Security Vulnerabilities found with and without security tools

This experiment is conducted to assess the effectiveness of the framework in detecting security vulnerabilities in the infrastructure. The vulnerabilities found by the pipeline will be compared against the vulnerabilities identified through manual inspection.

In order to conduct this experiment, first an IaC script with vulnerabilities had to be written. For this, IAM roles were configured with higher levels of privilege and permission, i.e "*" was used as a wildcard instead of specifying resources. Along with this, the S3 bucket was made publicly accessible and unencrypted. A secret was introduced which was hardcoded into the terraform script. API gateway was deployed with authorization set to "NONE".

When manually deployed using the terraform apply, some of these vulnerabilities were caught by Terraform itself such as the IAM role being of higher privilege and the S3 bucket with access control list set to none. However, the infrastructure was provisioned with the remaining vulnerabilities in place. A time consuming, manual

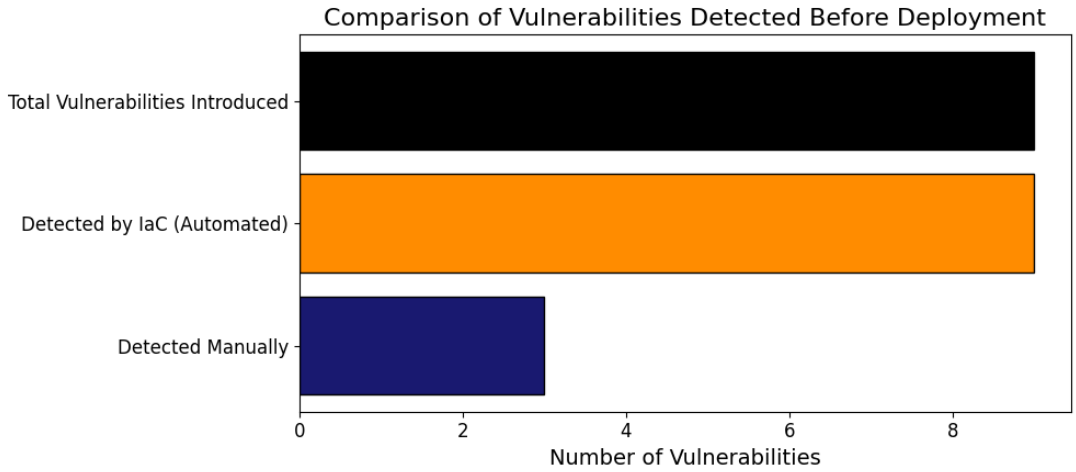

Figure 7: Comparision of Detection of Vulnerabilities in Automated and Manual deployment[Exp 3]

review was conducted using AWS Management Console and Trusted Advisor to find some of the remaining vulnerabilities.

When the same script was executed using the Jenkins pipeline, the Trivy scan picked up multiple vulnerabilities and the pipeline did not proceed till the vulnerabilities were fixed. This process was much faster when compared to the manual process, and some additional CVEs related to the python packages were detected. SonarCloud also picked up nuanced issues like exposure of public data and environment variables.

The security tools were able to find all the vulnerabilities that were intentionally added to the IaC Script, while manual inspection detected less than half of the total vulnerabilities that were introduced. However, this metric may vary depending on the expertise of the person conducting manual inspection and whether or not he uses any external tools. The results showed that automated security testing tools in the pipeline identified more vulnerabilities, in a shorter time, and more importantly, the application did not move to production until all the vulnerabilities were cleared, leading to a much more secure deployment of the application. The tools used in the pipeline proved invaluable for identifying vulnerabilities that would have otherwise been overlooked in manual reviews. Embedding all these tools in a CI/CD pipeline such as Jenkins ensures that the application will not be deployed until all the vulnerabilities and problems are fixed, reducing the risk of having insecure deployments in a production environment. However, in the testing it was also found that one of the vulnerabilities flagged by Trivy was a false positive, therefore it still required a manual validation before deployment.

5.3.4 Performance Impact of Security Checks

This experiment is conducted to evaluate the performance impact of integrating security checks into automated deployment pipelines. The deployment time is measured without security checks and with incremental additions of security checks in the pipeline. All times will be recorded three times and the value of the average will be considered for the experiment.

First, a basic CI/CD pipeline for deployment is set up in Jenkins, with only the three basic stages - Checkout stage to pull the code from GitHub, Plan stage to execute "terraform plan" and preview the infrastructure changes, Apply stage to finally deploy the infrastructure on AWS. The time taken for this basic deployment is recorded. Then, security checks are added incrementally. First, the SonarCloud scan stage is added. SonarCloud is integrated back into the pipeline to analyze code quality and vulnerabilities in Terraform scripts. This deployment time is then measured. Next, a basic Trivy scan stage is added which directly scans the Terraform script for known CVEs and other vulnerabilities. The time taken for this is recorded as well. Finally, a Dockerization step is added before the Trivy scan. The entire application will be containerized before Trivy scans it. In this way, Trivy can perform a more comprehensive scan of the environment where the application will run and all its dependencies. The time taken for this deployment is recorded as well.

As expected, the deployment time will increase as additional security checks are added, showing that adding more security checks increases the deployment time, as each step requires more compute time for processing. This experiment brings to the forefront the tradeoff of security and speed of deployment. It is always beneficial to have as much security checks as possible, however, in large scale deployments when each additional check could mean a few extra hours delay in deployment, a choice must be made between security and the speed of deployment.

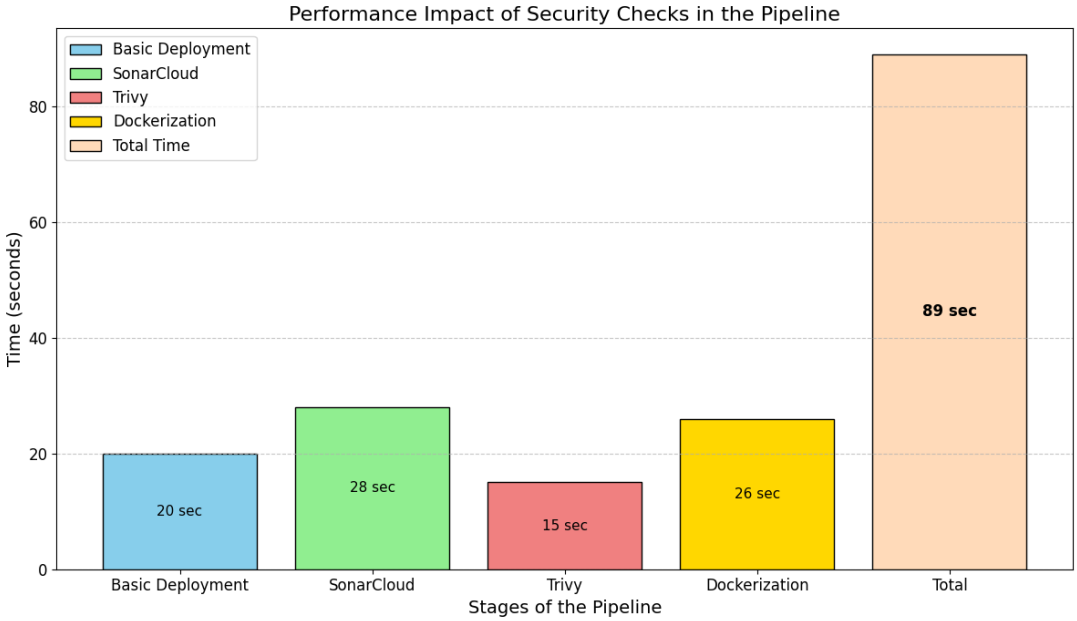

Figure 8: Performance Impact of the Security Checks[Exp 4]

There is also the consideration that even though there is a big initial delay in deployment due to these security checks, it can save considerable time later on in the process if any vulnerabilities have to be scanned for manually. The graph shown in fig 8 shows the performance impact of each tool individually, which is a helpful metric to take a decision on which tools to include in a framework, in case all of them cannot be added, and deployment time is the deciding factor.

6 Results

The experiments that were conducted show the stark differences between using Infrastructure as Code tools vs manual methods of serverless deployment. These results aim to answer the reesearch question i.e. what impact leveraging Infrastructure as Code tools has on the security of serverless deployments and configurations? This results section highlights the key findings from the experiments and showcase the advantages and challenges of using an IaC based approach for deployment.

The main advantage is the efficiency and time savings for deployment. Deploying a serverless application manually involves repetitive tasks like creating resources and manually configuring and assigning permissions and policies for each resource. From the experiments, the average deployment time was found to be around 30 minutes for a simple application, with variance based on the user's familiarity of the AWS Management console and other services that are being used. Even after deployment, debugging issues consumed a lot of time. In contrast, using Terraform scripts for deployment in a CI/CD pipeline reduced the time taken to 4 minutes after the initial setup is done. The security tools that were used greatly assisted in debugging by flagging all the misconfigurations and security vulnerabilities that were found. This brings us to the second experiment which showcased the effectiveness of IaC tools in mitigating configuration errors. Manual deployment requires humans to configure the infrastructure, and humans are prone to making mistakes like allowing higher levels of privilege to services which dont need it. These kind of issues get exponentially harder to detect as the scale of the deployment becomes bigger, and the onus is on the expertise of the user to detect such errors. Hashicorp's Terraform flagged some of these issues in the planning phase itself, where unused and overprivileged IAM roles were alerted to the user. Since Terraform cannot proceed until these alerts are resolved, it ensured that any potential vulnerabilities were addressed before they reached the live production environment.

The third experiment demonstrated the importance of having automated security tests for IaC deployments. Without security tools, manual review of the infrastructure is the only viable option to find vulnerabilities related to the infrastructure. Manual review is a tedious process which can consist of periodic code reviews and manual testing. However, this ultimately depends on the expertise and knowledge of the user to find these issues, and it is prone to human error. Such shortcomings can be remedied by using Security tools like Trivy and SonarCloud, which were able to detect vulnerabilities like hardcoded secrets in Terraform and CVEs in packages used by the application. The Jenkins pipeline terminated the deployment until all the vulnerabilities were deployed, ensuring

that the deployment is secure before it moves to production. The final experiment demonstrated the importance of having a good balance between security and deployment speed. The addition of security checks incrementally to the pipeline increased the deployment time proportionally. While the increase was still minimal due to the size of the application, it should be noted that these times will be much greater for a large scale deployment. Even if the time taken for deployment is a lot, it outweighs the time required for a manual vulnerability assessment later in the development cycle. The duty is deferred to the developer or company to decide if the investment in security early on is worth the longer deployment times.

The results of these experiments is corroborates the results found by Igwe (2024) who conducted similar experiments for general IaC deployments. The same results are seen to hold true for secure serverless deployments as well. The lack of standardization in IaC means that there are no guidelines that are set in stone for IaC implementations. However, over the course of this research undertaking, a few best practices were garnered and are listed below:

- Automate as much as you can: In large scale deployments, the lower the chances of human involvement, the more secure the pipeline will be. This also includes having automated security checks to continuously monitor the deployments.
- Use CI/CD pipelines: CI/CD pipeline software such as Jenkins can be adopted to automate the entire deployment process of serverless applications. Pipelines significantly reduce the manual inputs required and the time taken for deployments. Another factor in this is the use of containerization. Having a docker image of the application protects it in the bubble of the container, which can be secured more easily.
- Use Real Time Monitoring Systems: After the deployment is done, the threats to security do not stop. It is always important to have a continuous live monitoring system like Datadog to monitor a serverless deployment, which will ensure quick responses in case of any threat or outage.

7 Conclusion and Future Work

A survey by Kaseya found that 89% of IT professionals considered humans as the weakest link in the security of a system (Security Today, 2024). Taking this as the crux of the issue, this research focused on automating as much of the deployment process as possible using Infrastructure as Code tools along with a CI/CD pipeline and consequently bolstered the security of the entire deployment. Even industry leading organizations like Spotify use Infrastructure as code practices to make everything declarative in code (Jackson, 2024), thereby making it much easier for developers to focus on developing good software without being bogged down by infrastructure management. Studies have shown that integrating IaC tools within a CI/CD pipeline decreased the number of deployment errors by more than 65% and increased the speed of deployment by 50% (Kief and Portman, 2020). The experiments conducted in this research corroborated these same findings to an extent, cementing the fact that IaC tools and CI/CD pipelines go hand in hand when the focus is on security of deployments.

In 2020, JP Morgan Chase, a leading financial institution inducted Hashicorp formally inducted into the JP Morgan Chase Hall of Innovation to show recognition to the importance and business value that Hashicorp and Terraform have brought to their organization(Lehman, 2020). The organization praised Terraform for helping to standardize their public and private infrastructure provisioning for their multi cloud strategy. Building from this point, the findings from this research show that automating the serverless deployments using IaC is not just a convenience software, but is becoming more of a necessity in order to build scalable and repeatable serverless applications.

This research also established some best practices for securing serverless deployments, including the use of pipelines, containers and other automation technologies. There is scope in the research for further testing. Further research can be done focusing on comparing Terraform with other IaC tools like AWS CloudFormation. There can be more complex experiments set up to gain more nuanced insights into the benefits of IaC and automation in serverless deployments. Future research can also explore advanced use cases like optimizing IaC workflows in a hybrid or multi-cloud deployment. There is also scope to conduct more experiments on the security threats faced after deployment, in which case tools like Datadog and Falco can stand out as a bastion of security. The primary goal of the research was to assess the impact that IaC and automation tools have in the security of serverless deployments. The research addresses this goal by demonstrating how IaC tools embedded in a CI/CD pipeline reduces manual errors while improving consistency of deployment. The experiments validated that tools like Terraform not only improve the security but also the overall efficiency of serverless deployments. Furthermore, the findings from this study show the importance of using automation technologies in order to mitigate risks caused by human error. Organizations with large scale serverless deployments can use frameworks like this to optimize their deployments.

References

AWS (2024). *What Is AWS CloudTrail? - AWS CloudTrail*. Accessed: 2024-11-21. URL: https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-user-guide.html.

Bhatt, Ankit, Sachin Sharma, and Shuchi Bhadula (2024). "Security Issues in Serverless Cloud Computing Architectures". In: *2024 IEEE International Conference on Computing, Power and Communication Technologies (IC2PCT)*. Vol. 5. IEEE, pp. 39–43.

Chiari, Michele et al. (2024). "DOML: a new modelling approach to infrastructure-as-code". In: *Information Systems*, p. 102422.

Datadog (2023). *The State of Serverless*. https://www.datadoghq.com/state-of-serverless/. Accessed: 2024-07-31.

Drosos, Georgios-Petros et al. (2024). "When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2, pp. 2490–2520.

Igwe, Hephzibah (2024). "The significance of automating the integration of security and infrastructure as code in software development life cycle". PhD thesis. Purdue University.

Jackson, J. (2024). "PlatformCon: How Spotify Manages Infrastructure with GitOps". In: *The New Stack*. Online; accessed 2024-12-06. URL: https://thenewstack.io/platformcon-how-spotify-manages-infrastructure-with-gitops/.

Kamath, Shreyas, Shankar Vignesh, G Darshan, et al. (2023). "Revolutionizing cloud infrastructure management: Streamlined provisioning and monitoring with automated tools and user-friendly frontend interface". In: *2023 3rd International Conference on Intelligent Technologies (CONIT)*. IEEE, pp. 1–6.

Ketonen, Teemu (2024). "Strategies and challenges in cloud-to-cloud migration using infrastructure as code". In.

Kief, M and J Portman (2020). *Terraform: Up & Running: Writing Infrastructure as Code*.

Kumara, Indika et al. (2021). "The do's and don'ts of infrastructure code: A systematic gray literature review". In: *Information and Software Technology* 137, p. 106593.

Lehman, K. (2020). *HashiCorp Inducted Into JPMorgan Chase Hall of Innovation*. https://www.hashicorp.com/blog/hashicorp-inducted-into-jpmorgan-chase-hall-of-innovation. Accessed: 2024-12-06.

Li, Xing, Xue Leng, and Yan Chen (2022). "Securing serverless computing: Challenges, solutions, and opportunities". In: *IEEE Network* 37.2, pp. 166–173.

Marin, Eduard, Diego Perino, and Roberto Di Pietro (2022). "Serverless computing: a security perspective". In: *Journal of Cloud Computing* 11.1, p. 69.

Moghimi, Arshia (2024). "Automating resource and security configuration of serverless applications". PhD thesis. University of British Columbia.

Nastic, Stefan (2024). "Self-Provisioning Infrastructures for the Next Generation Serverless Computing". In: *SN Computer Science* 5.6, p. 678.

Nedeltcheva, Galia Novakova et al. (2023). "Challenges towards modeling and generating infrastructure-as-code". In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pp. 189–193.

Ntentos, Evangelos et al. (2024). "On the Understandability of Design-Level Security Practices in Infrastructure-as-Code Scripts and Deployment Architectures". In: *ACM Transactions on Software Engineering and Methodology*.

OWASP (2024). *OWASP Top Ten*. Available at: https://owasp.org/www-project-top-ten/. URL: https://owasp.org/www-project-top-ten/.

Raith, Philipp, Stefan Nastic, and Schahram Dustdar (2023). "Serverless edge computing—where we are and what lies ahead". In: *IEEE Internet Computing* 27.3, pp. 50–64.

Richter, Felix (2024). *Infographic: Amazon Dominates Public Cloud Market. Statista Infographics*. Accessed: 2024-11-21. URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

Sapkota, Dinesh (2023). "A Framework of DevSecOps for Software Development Teams". PhD thesis. University of Turku.

Security Today (2024). *Survey Finds Human Error is Cybersecurity Weakest Link*. [Accessed 6 Dec. 2024]. URL: https://securitytoday.com/Articles/2024/10/21/Survey-Finds-Human-Error-is-Cybersecurity-Weakest-Link.aspx.

Shuai, Zhang et al. (2024). "ATSSC: An attack tolerant system in serverless computing". In: *China Communications* 21.6, pp. 192–205.

Sokolowski, Daniel, David Spielmann, and Guido Salvaneschi (2024). "Automated Infrastructure as Code Program Testing". In: *IEEE Transactions on Software Engineering* 50.6, pp. 1585–1599. DOI: 10.1109/TSE.2024.3393070.

Waswani, N. (2020). *Serverless Architecture Patterns in AWS*. Accessed: 2024-11-30. URL: https://waswani.medium.com/serverless-architecture-patterns-in-aws-edeab0e46a32.

Westman, Roope (2022). "Automating a small-scale cloud environment". In.