The American Journal of
Engineering and Technology

ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 30-36
DOI 10.37547/tajet/Volume07Issue02-06

Check for updates

Building a fully automated serverless deployment pipeline with Aws lambda, terraform, and GITHUB actions

OPEN ACCESS

SUBMITED 16 December 2024
ACCEPTED 18 January 2025
PUBLISHED 20 February 2025
VOLUME Vol.07 Issue02 2025

CITATION

Diana Kutsa. (2025). Building a fully automated serverless deployment pipeline with Aws lambda, terraform, and github actions. The American Journal of Engineering and Technology, 7(02), 30–36. https://doi.org/10.37547/tajet/Volume07Issue02-06

Diana Kutsa

Bachelor of Management in Ternopil National Economic University, Crystal Lake IL, USA

Abstract: The paper discusses the process of building a fully automated pipeline for serverless deployment using AWS Lambda, Terraform and GitHub Actions. These technologies allow developers to create infrastructure without server management, manage it as code, and automate CI/CD processes. The article discusses the basic principles of using AWS Lambda to perform functions in a serverless architecture, Terraform for declarative infrastructure management, and GitHub Actions for automating the deployment process. The configuration steps are described, including creating and configuring IAM roles, connecting via API Gateway, and monitoring using AWS CloudWatch. The main focus is on the automatic scalability and flexibility of such solutions, as well as problems related to debugging and testing. The work highlights the benefits of integrating these tools to improve DevOps processes and accelerate application development.

Keywords: AWS Lambda, Terraform, GitHub Actions, serverless deployment, automation, CI/CD, DevOps, scalability, infrastructure as code.

Introduction: Modern IT infrastructures are rapidly evolving, and one of the key trends in this field is the shift toward serverless technologies. Serverless architectures allow developers to focus on writing code, eliminating the need to manage physical or virtual servers. This leads to significant reductions in infrastructure costs, while increasing the flexibility and scalability of applications. A crucial aspect of using such technologies is the ability to automate deployment and application management processes, which is particularly relevant given the ever-increasing demands for speed and quality in development.

One of the most popular platforms for implementing serverless solutions is AWS Lambda, which enables code execution without the need to manage servers. Combined with infrastructure-as-code tools like Terraform and CI/CD automation tools such as GitHub Actions, developers can build a fully automated deployment pipeline. This allows for the automation of not only resource creation and management but also testing and deployment, thereby enhancing the efficiency of DevOps processes.

The relevance of this topic is driven by the growing need to automate application development and deployment processes. The integration of AWS Lambda, Terraform, and GitHub Actions represents a powerful solution that minimizes manual operations, eliminates risks associated with human error, and accelerates the implementation of changes in applications. These technologies open new possibilities for building efficient, flexible, and scalable solutions that can quickly adapt to market and technology changes.

The purpose of this work is to explore the possibilities of building a fully automated pipeline for serverless deployment using AWS Lambda, Terraform, and GitHub Actions. The work aims to demonstrate the key stages and processes involved in creating such a pipeline, as well as to identify the advantages and potential challenges in its implementation.

1. Principles of Serverless Deployment with AWS Lambda

Serverless Framework is an efficient tool designed to simplify the process of deploying and managing applications in serverless architectures across various cloud providers, such as Amazon Web Services (AWS). To begin, AWS account credentials must be input into the AWS CLI configuration, enabling its use in conjunction with Serverless Framework for resource deployment. The following command can be used to create the configuration file:

```
cat <<EOF > ~/.aws/credentials
  [default]
  aws_access_key_id = <YOUR_ACCESS_KEY>
  aws_secret_access_key = <YOUR_SECRET_KEY>
EOF
```

Similarly, a configuration file for the region and output format can be created:

```
cat <<EOF > ~/.aws/config
  [default]
  region = eu-west-1
  output = json
EOF
```

In turn, Serverless Framework requires the presence of an IAM role to deploy resources on AWS. The following command creates this role:

```
aws iam create-role --role-name serverlessLabs --assume-role-policy-document '{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Principal": {
    "Service": "lambda.amazonaws.com"
   },
   "Action": "sts:AssumeRole"
  }
  ]
}'
```

Next, the AWSLambdaBasicExecutionRole policy must be attached to the created role:

```
aws iam attach-role-policy --role-name serverlessLabs --policy-arn
arn:aws:iam::aws:policy/AWSLambda_FullAccess
```

To verify the successful creation of the role, the following command can be used:

```
aws iam get-role --role-name serverlessLabs
```

The final project will represent a Python function deployed in AWS Lambda, connected to API Gateway and monitored via CloudWatch. The Serverless Framework can be installed with the following command:

```
npm install -g serverless
```

For convenience in organizing the project, a `functions` folder can be created to store the function:

```
mkdir functions
touch functions/__init__.py
```

This code defines a simple function that returns an HTTP response with a status code of 200 and a message in the response body. To define the Serverless configuration, the `serverless.yaml` file must be opened and the key parameters for deploying the function configured:

```
service: serverless-lab
provider:
  name: aws
  runtime: python3.7
  region: eu-west-1
  timeout: 10
  role: arn:aws:iam::155318317806:role/serverlessLabs
  memorySize: 512
functions:
  first_function:
    handler: handler.first_function
    events:
      - http:
          path: first
          method: get
```

This configuration defines the service, provider (AWS), function, and the parameters for invoking it through API Gateway. The API can be tested using a browser or tools such as cURL or Postman [1]. For example:

```
curl https://your_address.execute-api.region.amazonaws.com/dev/first
```

Serverless web applications offer several significant advantages over traditional server architectures or cloud infrastructure. Such systems provide automatic scalability, flexibility, and the ability to rapidly develop and release products. Table 1 presents the main advantages and challenges of these web applications.

Table 1. Advantages and challenges of serverless web applications [2].

Aspect	**Description**
Advantages of serverless web applications	
Automatic scalability	Serverless architectures automatically adapt to changes in load, including an increase in users and requests. This allows handling high loads, where traditional server solutions might become overloaded.
Flexibility in deployment	The lack of infrastructure requirements allows developers to quickly deploy applications and release updates. This accelerates innovation processes and enables adaptation to changing project requirements.
Fast time to market	Serverless technologies reduce the time for development and product release, enabling the phased upload and update of individual functions without stopping the entire application. This simplifies updates and bug fixes, ensuring continuous operation.
Challenges of serverless web applications	
Testing and debugging difficulties	Testing and debugging serverless applications are complicated due to the distributed nature of the application and the lack of centralized control over processes. This makes it harder to identify and fix errors, reducing transparency in the system's internal operations.
Process duration limitations	Serverless providers charge based on function execution time, making long-running operations less economically viable.
Performance impact	Serverless applications can experience delays during the initial launch of functions. Infrequent function usage may lead to increased loading times, although regular activity keeps the code ready for quick execution.
Vendor lock-in	The use of serverless technologies increases dependency on a specific provider, as each platform offers unique features and interaction methods. Switching providers can be challenging.
Scaling serverless applications	The main difficulties arise from the unpredictability of requests, making real-time auto-scaling configuration difficult. Additionally, serverless technologies limit control over resources, leading to challenges in ensuring stable performance under high loads.
Connection limits	To manage scalability, solutions such as Max Instances, Cloud Tasks for managing task execution speed, or Redis for rate-limiting requests are used.

Thus, serverless architectures offer high flexibility and scalability but require careful attention to testing, debugging, and performance management.

2. Infrastructure as Code Using Terraform

Terraform is an infrastructure management tool developed by HashiCorp, based on the concept of "infrastructure as code." It allows defining and describing resources and infrastructure elements in human-readable declarative configuration files, and efficiently manages their lifecycle. Terraform offers several advantages over manual infrastructure administration:

- Terraform can manage resources across multiple cloud platforms.

- The Terraform configuration language is intuitive and simplifies the process of writing infrastructure code.

- Terraform’s state mechanism allows tracking changes to infrastructure resources throughout all deployment cycles.

- Configurations can be stored in version control systems, making collaboration on infrastructure easier.

Terraform uses plugins, known as providers, to interact with various cloud platforms and services through APIs. Currently, over a thousand providers have been developed by the community and HashiCorp for platforms such as Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, and others. These plugins enable management of a wide range of resources, including compute power, networks, and monitoring services. The Terraform registry contains all available providers, and if a needed one is missing, custom providers can be created.

Each Terraform provider defines individual resources, such as virtual machines or networks, and these resources can be combined into modules for reuse. All resource management is handled through a unified process and language. Terraform uses a declarative approach to configuration: one describes the desired state of the infrastructure, rather than the sequence of steps to achieve that state, as in traditional programming languages. Terraform providers automatically compute dependencies between resources, ensuring that their deployment or deletion occurs in the correct sequence.

The infrastructure deployment process involves several steps:

- Defining the infrastructure required for the project.

- Writing a configuration that describes this infrastructure.

- Initializing, during which Terraform downloads the necessary plugins.

- Planning changes with a preview of the modifications that will be made to the infrastructure.

- Applying these changes to update the infrastructure according to the configuration [3].

Terraform allows the same infrastructure to be described in various ways. Consider the following example, where an EC2 instance with an attached EBS volume is described differently, but the result remains the same:

```
resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
}

resource "aws_ebs_volume" "example-volume" {
  availability_zone = "${aws_instance.example.availability_zone}"
  type              = "gp2"
  size              = 100
}

resource "aws_volume_attachment" "example-volume-attachment" {
  device_name = "/dev/xvdb"
  instance_id = "${aws_instance.example.id}"
  volume_id   = "${aws_ebs_volume.example-volume.id}"
}
```

In this example, the EBS volume is presented as a separate resource, which allows for more flexible management of volumes, making it possible to add or remove them without needing to modify the EC2 instance [4]. Below are the main advantages that make Terraform a preferred choice:

- Syntax: Terraform's HCL syntax is more readable and concise compared to JSON in ARM templates, and it includes functions for variable interpolation.

- Modules and structure: Terraform offers simple and intuitive code modularization, making it easier to work with compared to linked templates in ARM.

- Dependency management: Terraform automatically recognizes dependencies between resources, ensuring changes are applied in the correct sequence. In ARM, these dependencies must be explicitly specified manually.

- Change planning: One of Terraform's key features is the ability to preview changes during the planning phase, providing transparency and helping to avoid unexpected errors. ARM implements a similar feature through the "what-if" command, but it is not an essential part of the deployment process.

- Cross-platform compatibility: Terraform integrates with most public clouds, including Kubernetes and other platforms, making it a flexible tool for building hybrid infrastructures.

However, Terraform also has some limitations:

- Delay in supporting new features: Although Terraform for Azure is actively supported, new feature support may appear with some delay. In cases where new features are required, ARM templates can be used through Terraform resources.

- Lack of integration with Azure Portal: Unlike ARM, Terraform does not provide the ability to create resources through the Azure portal and then download templates.

- State management: Terraform requires managing the state of controlled resources, which may introduce additional security and high-availability requirements for storage.

Terraform Enterprise (TFE) and its SaaS version, Terraform Cloud, offer additional features compared to the basic open-source product, such as remote execution and state management. However, most of these features can also be implemented without the Enterprise version through integration with CI/CD platforms [5].

This rethinking positions Terraform strongly in the corporate environment, providing powerful tools for infrastructure management and deployment.

3. CI/CD Automation with GitHub Actions

GitHub Actions is an advanced tool for automating various stages of software development, such as continuous integration (CI) and continuous deployment (CD).

Continuous integration (CI) is a practice where all code changes made to a shared repository are automatically tested for errors using automated tests. This allows developers to quickly identify and fix issues, improving code quality and speeding up development.

Continuous deployment (CD) is a logical extension of CI, where changes from the repository are automatically deployed to production servers. This automation significantly accelerates the delivery of new features and bug fixes.

GitHub Actions plays a key role in automating these processes, enabling the creation and management of CI/CD pipelines directly within GitHub repositories. This platform allows for the automation of tasks such as code testing, building, compliance checks, and application deployment.

The principles of CI/CD go beyond mere tools—it is a development philosophy aimed at minimizing risks when implementing changes in software. The more frequently changes are integrated and deployed, the easier it is to manage the process and maintain product stability.

One of the key advantages of GitHub Actions is its flexibility. This tool allows the configuration of workflows with multiple steps, ranging from simple test execution to complex multi-stage deployments. Integration with GitHub also simplifies CI/CD process management, making them more transparent and user-friendly [6].

Thus, the implementation of GitHub Actions in projects significantly optimizes development, making it more predictable and efficient, which improves both the quality and speed of development teams' work.

CONCLUSION

Building a fully automated pipeline for serverless deployment using AWS Lambda, Terraform, and GitHub Actions is an effective way to accelerate the development and deployment of applications. The integration of these tools significantly enhances the flexibility and scalability of the architecture, allowing developers to quickly adapt to changing requirements and efficiently release updates. Despite some challenges, such as debugging and testing distributed functions, this approach greatly simplifies DevOps processes and improves team productivity. Further improvements could include optimizing the pipeline architecture, utilizing additional tools for monitoring and infrastructure management, and refining automated testing processes.

REFERENCES

Nestorov A. M. et al. Performance evaluation of data-centric workloads in serverless environments //2021 IEEE 14th International Conference on Cloud Computing (CLOUD). – IEEE, 2021. – pp. 491-496.

Choudhary B. et al. Case Study: use of AWS lambda for building a serverless chat application //Proceeding of International Conference on Computational Science and Applications: ICCSA 2019. – Springer Singapore, 2020. – pp. 237-244.

Begoug M., Chouchen M., Ouni A. TerraMetrics: An Open Source Tool for Infrastructure-as-Code (IaC) Quality Metrics in Terraform //Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. – 2024. – P. 450-454.

Pandya S., Guha Thakurta R. Hands-on infrastructure as code with hashicorp terraform //Introduction to Infrastructure as Code: A Brief Guide to the Future of DevOps. – Berkeley, CA: Apress, 2022. – pp. 99-133.

Kumar M. et al. Infrastructure as code (IAC): insights on various platforms // Sentiment Analysis and Deep Learning: Proceedings of ICSADL 2022. – Singapore: Springer Nature Singapore, 2023. – pp. 439-449.

Decan A. et al. On the use of github actions in software development repositories //2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). – IEEE, 2022. – pp. 235-245.