

Optimizing AWS Resource Provisioning Time through Thread-Level Parallelism and Terraform

MSc Research Project
Cloud Computing

Sanket Patil
Student ID: X21229139

School of Computing
National College of Ireland

Supervisor: Sean Heeney

National College of Ireland
Project Submission Sheet
School of Computing



Student Name:	Sanket Patil
Student ID:	X21229139
Programme:	Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Sean Heeney
Submission Due Date:	14/12/2023
Project Title:	Optimizing AWS Resource Provisioning Time through Thread-Level Parallelism and Terraform
Word Count:	6562
Page Count:	21

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:	
Date:	14th December 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
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.	<input type="checkbox"/>

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):	

Optimizing AWS Resource Provisioning Time through Thread-Level Parallelism and Terraform

Sanket Patil
X21229139

Abstract

This research addresses the critical challenge of optimizing deployment time for AWS resources by introducing a novel strategy that combines thread-level parallelism and Infrastructure-as-Code (IaC) automation. Thread-level parallelism is a technique that involves dividing a program into smaller tasks, called threads, which can be executed concurrently. The core of the study revolves around the use of Python's library to implement thread-level parallelism for the simultaneous execution of multiple Terraform scripts responsible for provisioning key AWS services like S3, Elastic Beanstalk, and RDS. By using thread-level parallelism, multiple Terraform scripts can be executed simultaneously, reducing the overall deployment time.

In this research, terraform is used as the IaC tool to automate the provisioning of AWS resources. Terraform scripts are written in a declarative language that describes the desired state of the infrastructure. By automating the provisioning process, the research aims to streamline and optimize the deployment of AWS resources. Jenkins, functioning as the deployment server, retrieves these scripts and Python code from the GitHub repository, orchestrating the provisioning of AWS resources based on the pre-defined configuration. The integration of thread-level parallelism enhances deployment efficiency, enabling concurrent execution of resource provisioning tasks for faster and more scalable deployments. To validate the effectiveness of this proposed system, a web application is developed using the Django framework in Python. This application leverages AWS resources provisioned through the optimized deployment strategy, showcasing the practical implications of both thread-level parallelism and infrastructure-as-Code automation. The seamless integration of these technologies not only streamlines the deployment process but also ensures the reliability and consistency of the deployed infrastructure. .

The results of the study show a significant difference in the time required for AWS (Amazon Web Services) resource provisioning when parallel execution is used instead of sequential execution.

This project serves as a valuable demonstration of modern deployment practices, emphasizing the importance of automation and parallelism in achieving efficient and scalable AWS resource provisioning for web applications. The presented system offers a practical and replicable approach for teams seeking to enhance their deployment workflows and maximize the benefits of cloud infrastructure.

1 Introduction

In the rapidly evolving landscape of cloud computing, organizations are increasingly leveraging the flexibility and scalability offered by platforms such as Amazon Web Services (AWS) to meet their computational demands. As enterprises embrace cloud-native architectures, the efficient provisioning and deployment of AWS resources becomes pivotal for achieving optimal performance and resource utilization. This is particularly crucial in dynamic environments where the ability to scale resources swiftly can make the difference between meeting and exceeding business objectives. The rapid and efficient deployment of resources is paramount for organizations seeking to leverage the benefits of platforms like Amazon Web Services (AWS). As cloud infrastructures grow in complexity, provisioning various AWS services such as S3, Elastic Beanstalk, and RDS demands an approach that not only ensures accuracy in resource creation but also minimizes deployment times. The deployment time of AWS resources is a significant issue that cloud practitioners must deal with since it directly affects an organization's overall agility. Sequential execution is a common feature of traditional resource provisioning techniques, which may cause bottlenecks and extend deployment times. This paper, recognizing the need for increased efficiency, presents a novel method to integrate Infrastructure-as-Code (IaC) automation and Thread-Level Parallelism (TLP) to optimize the deployment time of AWS resource provisioning. Optimize AWS resource provisioning deployment time through the integration of Thread-Level Parallelism (TLP) and Infrastructure-as-Code (IaC) automation. Complementing TLP, Infrastructure-as-Code automation emerges as a fundamental element in achieving consistency, repeatability, and manageability in cloud resource provisioning. By representing infrastructure configurations as code, IaC enables automated, version-controlled, and collaborative management of AWS resources. This paper explores the constructive collaboration between TLP and IaC, demonstrating how their integration can lead to a streamlined and accelerated deployment process.

As organizations navigate the intricacies of the cloud, the proposed optimization strategy holds promise for not only expediting resource deployment but also enhancing resource utilization and cost-effectiveness. By harnessing the power of parallelism and embracing automation through Infrastructure-as-Code, enterprises can position themselves to meet the demands of dynamic business environments with agility and efficiency. This paper delves into the theoretical underpinnings, practical implementation, and potential benefits of this innovative approach, contributing valuable insights to the evolving landscape of cloud infrastructure management.

The research coordinates the deployment of AWS services, such as S3, Elastic Beanstalk, and RDS, using Terraform, an infrastructure-as-code scripting language, to automate resource deployment. Each AWS resource is arranged into distinct workspaces, or directories, through a unique Terraform script. The research focuses on using Python to implement multi-threading and create a parallel script. This script makes use of the library's functions to run multiple threads simultaneously. The "workspace" variable, containing a list of Terraform scripts, serves as the basis for parallel execution. The `deploy_environment` function encapsulates Terraform commands like `'terraform init'` and `'terraform apply'`, supplemented by parameters such as `'-auto-approve'` to circumvent state lock conditions that might hinder concurrent execution.

1.1 Motivation

Traditional resource provisioning techniques often involve sequential execution as proposed in Ivanova et al. (2018), leading to bottlenecks and extended deployment times, especially as cloud infrastructures grow in complexity. Recognizing the need for increased efficiency in this context, the motivation for this research stems from the desire to address the challenges associated with the deployment time of AWS resources. The paper aims to explore innovative approaches that go beyond conventional methods, leveraging Infrastructure-as-Code (IaC) automation and Thread-Level Parallelism (TLP) to optimize AWS resource provisioning deployment time.

1.2 Research Question

How does the integration of Thread-Level Parallelism (TLP) and Infrastructure-as-Code (IaC) enhance the efficiency of cloud resource deployment, considering both theoretical foundations and practical implementation?

1.3 Report Structure

This paper proposed the parallel execution approach for executing the terraform scripts to provisioned AWS resources. Related work is described in section 2. The methodology of the study is detailed in section 3. Section 4 and section 5 describes Design specification and implementation of the proposed research work. Evaluation of the proposed research work is discussed in section 6. Whereas, section 7 describes Conclusion and future work.

2 Related Work

In modern IT infrastructure management, the ability to provision cloud resources rapidly and effectively is critical for organizations to scale their operations, satisfy customer needs, and optimize costs. This literature study focuses on the present state of research and techniques for optimizing the deployment time of AWS (Amazon Web Services) resources. It focuses specifically on the synergistic usage of thread-level parallelism and Infrastructure-as-Code (IaC) automation.

2.1 Review on Infrastructure-as-Code (IaC)

IaC has drawn a lot of attention since it can use machine-readable scripts to create and maintain infrastructure. These scripts can be versioned, allowing for easy tracking of changes and team cooperation. Organizations may achieve consistency and repeatability in their infrastructure deployments by leveraging IaC, lowering the risk of human error and guaranteeing that infrastructure is always provided and configured appropriately. Artac et al. (2017) conducted research on the benefits and use cases of Infrastructure-as-Code (IaC) in DevOps. The author discusses various advantages of IaC in the context of DevOps, such as versioning, which allows for debugging and back-tracking/version control. It also enables the use of infrastructure design patterns to swiftly assemble solutions, which are pre-defined solutions to common problems. Additionally, Model-driven engineering can also be utilized to directly derive IaC specifications from architectural

design and development models. Infrastructure-as-code is an important part of DevOps since it allows several essential principles that rely heavily on automation. On top of that, Research by Guerriero et al. (2019) highlights Adoption, Support, and Challenges of Infrastructure-as-Code. The report draws on data from 44 semi structured interviews with senior developers from a variety of firms. The outcomes of this study emphasize the need for additional research in the field of IaC. The tools for IaC support that are now available are rather restricted. According to the findings, there is a lack of tools and procedures that can successfully support the adoption and maintenance of IaC. The study intends to fill this void. The methodology of this research comprises analyzing (1) current development practice, (2) the advantages and drawbacks of available tools, and (3) problems when writing infrastructure code.

Technologies like virtualization, cloud computing, containers, server automation, and software-defined networking are made to make IT operations easier. The goal of these technologies is to improve infrastructure and handling of resources for businesses. Many firms have discovered, nevertheless, that using these technologies frequently results in a sprawl of unmanageable systems. This implies that these technologies may potentially increase complexity and manageability rather than simplify processes. Infrastructure as code can help with this. This notion entails utilizing code, as opposed to human methods, to manage infrastructure resources. Organizations may increase the efficiency and manageability of their infrastructure by automating the provisioning, configuration, and management processes with infrastructure as code. The study Morris (2016) describes how to apply the DevOps movement's tenets, procedures, and patterns to infrastructure as code in the cloud era. This study is broken up into three sections. The first section describes the platforms and tools used to create and configure infrastructure components. Understanding the functionalities and service models of dynamic infrastructure platforms is part of this. The research's second section is devoted to patterns for using the instruments and methods. It offers information on how to use infrastructure as code to upgrade servers that are already in operation, create server templates, and provide servers. The final section of the article discusses techniques for ensuring that infrastructure as code functions well in various settings.

Additionally, the study Ikeshita et al. (2017) included ways to shorten test durations and expenses when examining idempotence in target infrastructure code that incorporates script resources. Research includes earlier investigations that looked at various methods to determine whether or not the code used in IaC is idempotent. These investigations relied on two standard methods for determining idempotence in code. But this method can be cumbersome and unworkable, particularly in the case of intricate infrastructure configurations. The researcher in Ikeshita et al. (2017) suggests an effective way to verify idempotence in IaC code by combining testing and static verification methodologies. The goal of this study is to minimize the amount of test cases needed, especially for code that uses external scripts. The technique works well for cutting down on the amount of tests required to verify idempotence. The approach's usefulness is demonstrated in the article through tests, which also indicate that the method had an acceptable execution time and was successful in lowering the size of the test suite. According to research, this method provides a more useful and efficient way to verify idempotence in IaC code, assuring the infrastructure's stability and dependability.

The study report de Carvalho and Patricia Favacho de Araujo (2020) then examines several infrastructure management solutions and compares their respective performances. This study examines many technologies, including Terraform, Heat, CloudFormation, and

Cloudify Brikman (2022). The analysis’s findings show that only Cloudify and Terraform are appropriate for the particular needs outlined in this study among all the orchestrators that were taken into consideration. In this experiment, a Wordpress application’s blueprint is provisioned across many cloud service providers. The Wordpress application’s required resources, including virtual machines, network resources, security restrictions, and routing, are specified in the design. The study’s findings indicate that Terraform outperformed Cloudify in terms of speed throughout both the provisioning and deprovisioning phases. Terraform took an average of 4.7 minutes to furnish the test environment, whereas Cloudify took an average of 8.5 minutes. In addition, it was found that compared to Terraform, Cloudify used greater network resources. Terraform used an average of 1.3 gigabytes of network bandwidth to run its activities during the provisioning phase. However, Cloudify required nearly twice as much network traffic—2.3 gigabytes on average—to carry out the identical provisioning task. Disk activity is another criterion measured in the study, and it was found that Terraform had a modest average disk activity, gathering data during its operations totaling between 0.1 and 0.2 megabytes. Cloudify, on the other hand, accumulated data ranging from 1.6 to 2.1 megabytes and showed increased disk activity. The study concludes that Terraform is probably able to handle data more effectively than Cloudify since it has less disk activity and concentrates more on memory, which leads to a shorter execution time.

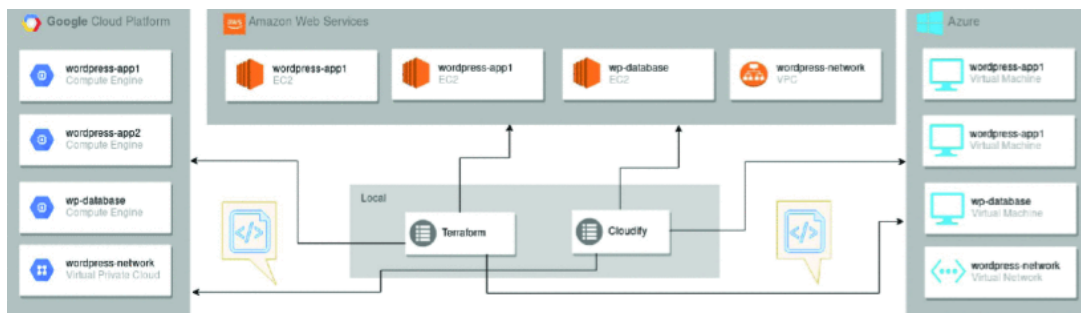


Figure 1: Deployment of multi cloud resources using terraform and cloudifyde Carvalho and Patricia Favacho de Araujo (2020)

2.2 Review on Thread-level Parallelism

The capacity of a computer system to run many threads at once is known as thread-level parallelism. TLP is centered on making advantage of numerous threads to improve a computer system’s overall performance. Computer systems can get increased throughput—the quantity of jobs finished in a given amount of time—by utilizing TLP. Researchers in article Aziz et al. (2021) reviewed multiprocessing and parallel processing in Python . Rather of addressing a huge issue sequentially, parallel and multiprocessing algorithms break down important numerical problems into smaller sections that may be tackled simultaneously on separate processors or cores. The computing process may be sped up considerably with this method. Python and C are two programming languages that handle it effectively. These languages are seen to be appropriate for ”heavy-duty” computing jobs as they come with the libraries and tools needed to carry out parallel algorithm implementation. Global interpreter lock (GIL), a Python feature that guarantees only one thread executes Python bytecode at a time, was discussed in Paper Aziz et al.

(2021). This indicates that real parallel execution of several threads is not supported to the fullest extent by Python’s default interpreter. Nonetheless, a wide range of Python tools and packages have been created throughout time to facilitate multiprocessing and parallel processing. These libraries allow for the effective parallel execution of jobs and offer remedies for the GIL’s constraints. The study focuses on Python packages designed to enable multiprocessing and parallel processing.

A different study Babuji et al. (2017) presents Parsl, a Python parallel scripting framework that aids in the creation and implementation of extensive, subject-specific workflows by academics. A software framework called Parsl enables parallel task execution within a Python script. Parsl launches asynchronous Apps in place of computing results synchronously (one after the other). The Dataflow Kernel, or DFK for short, is a Parsl component that controls how a script is executed. The DFK makes sure that the script’s regular functions are informed about the futures (AppFutures) and their dependencies. These functions will only run when all required outcomes are available, as their execution is contingent on the resolution of all dependent futures. This guarantees that the functions’ execution and the results’ availability are in sync. There are now three executors available in Parsl that may be used to run tasks concurrently. Threads, Swift/T, and IPython-Parallel are these executors. Standard Python code plus many Apps make up a Parsl script. Because Parsl Apps are entirely asynchronous, the outcome of an App invocation is not returned right away. Parsl returns an AppFuture rather than a result directly. An AppFuture is a construct that holds the exceptions, status, and actual outcome for that particular asynchronous function call. Parsl offers several ways to inspect the AppFuture, including status checking, stopping on completion, and result retrieval. Parsl uses the concurrent.futures package, which is built in Python, to manage the AppFutures in this Babuji et al. (2017) work.

Additionally, the author of article Kim et al. (2021) conducts a survey and performance test of Python-based parallel processing packages. These libraries provide ready-made code that may be used to accomplish specialized tasks like data analysis or machine learning. The ability to execute numerous tasks at once using the Python-based parallel processing library can greatly accelerate the execution process. Parallel processing, according to studies, is breaking a task up into smaller subtasks that may be run concurrently on several processors or cores, which cuts down on processing time overall. This methodology facilitates expedited development and seamless integration of disparate software components. Kim et al. (2021) suggested a study that assessed the performance of six Python parallel processing libraries, including Ipyparallel. The study’s goal was to see how successful parallel processing is at improving performance when compared to serial processing. The researchers used Python libraries to conduct both a practical test and a theoretical analysis of parallel processing; the testing revealed that parallel processing enhanced performance by around 71.6% when compared to serial processing. The study sought to demonstrate the advantages of parallel processing in terms of improved performance. It concludes that parallel processing can greatly improve efficiency and speed. Parallelism is the ability to perform many tasks at the same time.

The authors of the following paper Galvez et al. (2018) describe CharmPy, a parallel programming model and framework based on the Python programming language. CharmPy is a Python-based parallel programming language that works on top of Charm++’s C++ runtime. CharmPy is a high-level paradigm based on the concept of distributed migratable objects, which enables things to migrate between parallel processors. The authors of Galvez et al. (2018) show how to create a distributed parallel map function built

on the Master-Worker pattern, with support for asynchronous concurrent jobs, to demonstrate the simplicity of the CharmPy architecture. The authors also describe results from executing stencil code (a sort of method used in image processing) and molecular dynamics mini-apps (small programs mimicking molecular activity) written entirely in Python using CharmPy on the Blue Waters and Cori supercomputers. Whereas, the author of the following research study Chansup Byun (2022) employs pPython for parallel Python programming. pPython is a programming language that tries to give parallel computing capabilities while retaining the simplicity of Python programming. It accomplishes this by layering partitioned global array semantics (PGAS) on top of PythonMPI, a simple file-based messaging package. The SPMD (single program multiple data) paradigm of processing is used by pPython, which means that the same program is performed on several processors, but each processor operates on a distinct section of the data. The paper Chansup Byun (2022) compares pPython against pMatlab using the HPC Challenge benchmark suite and offers preliminary results. This report also discusses difficulties and future work.

2.3 Literature Review Summary

In summary, in this paper Artac et al. (2017), author discuss discusses the advantages of Infrastructure-as-Code (IaC) in DevOps, how to leverage Model-driven engineering to derive IaC specifications from architectural models, and how to employ infrastructure design patterns for rapid problem-solving. According to Guerriero et al. (2019), Infrastructure-as-Code (IaC) requires extra tools and processes to facilitate adoption and maintenance as the available solutions has limitations and difficulties associated with producing infrastructure code. In Morris (2016) study describes how to use DevOps ideas to infrastructure as code in the cloud era. It covers tools and platforms, infrastructure as code patterns, and methods for making sure the code works in various environments. Whereas, in Ikeshita et al. (2017), study propose an approach that reduces the amount of test cases required to validate idempotence in infrastructure code. The strategy works well for cutting down the execution time. The comparison of different infrastructure management solutions including Terraform, Heat, CloudFormation, and Cloudify discuss in de Carvalho and Patricia Favacho de Araujo (2020), outcome of this study shows, Terraform outperforms Cloudify in terms of execution time.

In Aziz et al. (2021), Researchers examined Python’s parallel and multiprocessing features. It also discusses limitations and alternative for parallel processing. In Babuji et al. (2017), study presents, Parsl – a python based parallel processing framework that enables parallel task execution using concurrent.futures package. According to a Kim et al. (2021), parallel processing outperformed serial processing by about 71.6% when six Python parallel processing libraries were evaluated. Whereas, in Galvez et al. (2018) and Chansup Byun (2022), study introduce charmPy and pPython as an another parallel Python programming languages. While pPython employs partitioned global array semantics and evaluates its performance against pMatlab, CharmPy uses a distributed parallel map function to highlight the simplicity of its architecture.

3 Methodology

The primary objective of this research is to enhance resource provisioning by optimizing time taken by the execution of terraform script with the help of thread-level parallelism.

In this project, Amazon Web Services (AWS) is used to host the resources deployed through the execution. The initial step of this methodology begins with creating the terraform script. It contains three different terraform script to deploy AWS RDS, Elastic Beanstalk and S3. These scripts are stored in different directories under the workspace directory. These directories will be accessed by the python script stored in the GitHub repository in order to fetch the terraform script.

This methodology's Terraform script for RDS service will generate a security group and a relational database service. The security group created by this script will allow incoming and outgoing traffic to the RDS database. Another Terraform script will generate an S3 bucket. This script will also apply S3 bucket policies to newly created buckets. In contrast, terraform script designed for AWS elastic beanstalk service would generate elastic beanstalk environment as well as beanstalk application. This script will also retrieve the web application zip file from the S3 bucket and deploy it to the application. Furthermore, resource deployment process also includes GitHub repository that used to store the terraform scripts along with the python script and Jenkins file. The structure of the GitHub repository is shown in the Figure 2. Jenkins is used in this project as a deployment server. Jenkins server is configured to access the Jenkins file stored in the GitHub repository. Jenkins file contains bat command that used to execute the python script.

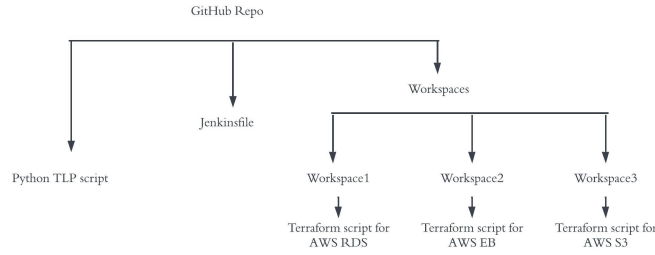


Figure 2: Architecture of the Git Repository

3.1 Execution Workflow

Whenever, pipeline created in Jenkins server will get execute, it will access the Jenkins file stored in the GitHub repository. Bat command stored in that file will run the python script. after executing the bat command python script will access the terraform script stored in the workspace directory and it will run terraform command on it. concurrent.futures library used in the python script will assign the execution of terraform script to individual thread to perform concurrent execution of resource deployment. As a result, this execution process will create three different AWS resources such as AWS RDS, Elastic Beanstalk and S3 in minimal time.

3.2 Demonstration Setup

The proposed work in this research is to verify and confirm the improvement in deployment time achieved through parallel deployment methods. This is done by evaluating performance metrics such as deployment time and comparing them with metrics obtained

from other deployment methods. For evaluation purpose, methodology proposed in this research work incorporates three different approaches for hosting the AWS services.

3.2.1 Traditional Approach

In the traditional approach, a python script will not be used to automate the execution of the terraform scripts; instead, each terraform script will be executed one by one using multiple Jenkins pipelines. This method does not guarantee concurrent execution of Terraform scripts.

3.2.2 Sequential Approach

The python programming language will be utilized in this technique to automate the execution process of terraform scripts. The python script built in this approach will fetch the terraform script and execute it sequentially. This method also does not ensure that the terraform scripts run concurrently.

3.2.3 Parallel Approach

This method employs Thread-Level Parallelism (TLP), which will be implemented using a Python script. The Python script used in this case will retrieve the Terraform script from the workspace directory and assign it to the separate thread for execution. This thread will run the Terraform script in parallel.

4 Design Specification

The research proposed on integration of thread-level parallelism (TLP) with Infrastructure-as-Code (IaC) is designed to optimize AWS resource provisioning by minimizing the execution time. This section outlines the design specifications for the proposed research work. This research includes various technical aspects required for the provisioning of AWS resources such as terraform, Jenkins, GitHub, etc. To implement Thread-level parallelism python 3.8 is incorporated in this research.

4.1 Proposed architecture:

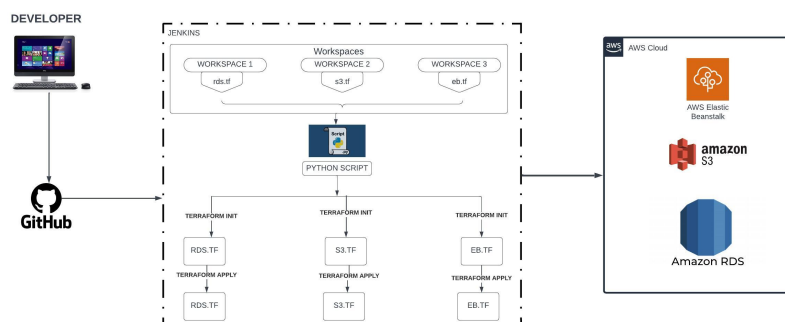


Figure 3: Proposed architecture design for enhance resource deployment

As shown in Figure 3, this research includes GitHub repository as a source control management. In proposed work GitHub repository is used to store the terraform script and python code. GitHub repository used in this proposed work contains three different terraform scripts that stored in three different directories under parent directory. Each terraform script is used to create three different AWS resources such as AWS RDS, Elastic Beanstalk and S3. GitHub repository also contains Jenkins file that used by the Jenkins deployment server to run the python script. Jenkins file stored in the GitHub contains bat command that trigger the python script when Jenkins run the new build. After successful build Jenkins will spin up three resources that is AWS RDS, Elastic Beanstalk and S3 in AWS subscription.

4.2 Python TLP Architecture

Main idea of this research work is focused on the Python script that used to implement thread-level parallelism(TLP). The python script divides into three logical section. First section of the script contains workspace variable with list data type. This variable stores the terraform workspace directories that stores terraform script. This section also contains max_workers variable which used to specify number of threads used during the execution. Second section of the script contains deploy_environment () function. This function contains terraform command along with additional parameter such as auto-approve, lock, etc. to avoid state lock condition while executing multiple terraform command simultaneously. Terraform commands include in this function are terraform init (used to initialize terraform script) and terraform apply (used to provision resources as per configuration). It also contains script_dir variable that reads path of terraform script specified in workspaces variable in first section. In final section of the script ThreadPoolExecutor method of concurrent.futures library is used which is the core component of the script used to implement thread-level parallelism. This method maps the each terraform script with terraform command and assign it to the individual thread for parallel execution

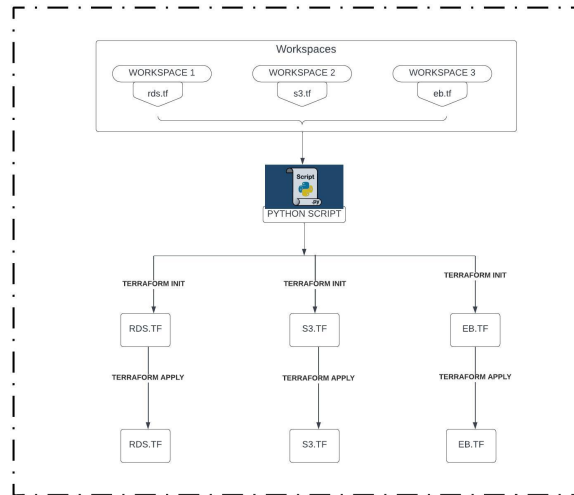


Figure 4: Python Thread-Level Parallelism Code Architecture

As shown in Figure 4, On execution of the script, python will run terraform init command on all terraform script and after successful completion of terraform init command it

will start executing terraform apply command on all terraform script. It will execute both command on all terraform script simultaneously by assigning execution of the terraform script to individual thread.

5 Implementation

Objective in this research is to optimize deployment time of AWS resource provisioning with the help of thread level parallelism. This can be done with the help of executing multiple thread simultaneously with the help ‘concurrent.futures’ library of python programming language. for the demonstration purpose, AWS services like S3, elastic beanstalk and RDS has been created. For provisioning of this resources terraform is used. Terraform is a scripting language also known as Infrastructure-as-a-Code that enabled automation in resource deployment. In this project, as shown in Figure 2, for every resource separate terraform script has been written and stored into the separate directory. These directories are renaming as workspace. So, each workspace contains separate terraform script. Each terraform script in this project stored AWS credentials that is required in order to deploy the resources using IaC method.

5.1 Thread-Level Parallelism (TLP) Implementation

To implement multi-threading with the help of thread-level parallelism, python programming language is used. By using python programming language, python TLP (Thread-level parallelism) script has been developed. This script contains a ‘workspaces’ variable with data type as a list that fetch and stored all terraform script. Whereas, ‘deploy_environment’ function included into script contains terraform command such as ‘terraform init’ and ‘terraform apply’ along with other parameter such as ‘-auto-approve’ and ‘-lock=false’ to avoid state lock condition in terraform. state lock condition in terraform restrict execution of multiple terraform script by locking the state of currently executing terraform script

The TLP script developed in this project also includes ‘max_workers’ parameter which used to define maximum number of concurrent threads to be utilized during the execution of deployment process. Number of workers are depending on the system capabilities.

Finally, ‘ThreadPoolExecutor’ and ‘as_completed’ function of ‘concurrent.futures’ library is used that enables parallel execution of the terraform script. This function maps terraform command with individual workspace and assign it to the separate thread. These threads execute terraform command on every workspace in parallel manner.

5.2 Interaction with Terraform

Proposed implementation includes terraform as a scripting language that used to automate deployment process of AWS resources such as RDS, Elastic Beanstalk and S3. Each scripts developed in this implementation includes multiple terraform module as per requirement. This section includes details of the all three terraform scripts proposed in this implementation.

5.2.1 Relational Database service with Terraform

In this script ‘aws_db_instance’ and ‘aws_security_group’ resource blocks of terraform is used. ‘aws_db_instance’ block used to create database instance. This block includes required and optional attribute such as allocated_storage, db_name, engine, engine_version, etc. In this implementation, this block is used to create MYSQL 5.7 database instance along with storage of 20GB, t3.micro class instance, username and password to access the database. On other hand, ‘aws_security_group’ block included in the script is used to create security group with name ‘parallel-mysql-sg1’ that allow network traffic on port 3306. Security group created in this resource block is configured in ‘vpc_security_group_ids’ attribute of ‘aws_db_instance’ resource block.

5.2.2 Elastic Beanstalk with Terraform

The script defined for elastic beanstalk in this implementation includes three resource blocks such as ‘aws_elastic_beanstalk_application’, ‘aws_elastic_beanstalk_environment’ and ‘aws_elastic_beanstalk_application_version’. In this implementation, ‘aws_elastic_beanstalk_application’ block is used to create elastic beanstalk application with name ‘parallel-research-ElasticBeanstalkApp’. Whereas, ‘aws_elastic_beanstalk_environment’ block is used to create beanstalk environment with name ‘parallel-research-Environment’. This blocks include other required parameter such as application name, solution stack name and version label. Last block includes in this script that is ‘aws_elastic_beanstalk_application_version’ is used to deploy web application on created elastic beanstalk environment. This block required attribute such as application name on which web application will be deployed, location and name of web application zip file.

5.2.3 S3 with Terraform

Final script included in this implementation is to provisioned AWS S3 resources which includes ‘aws_s3_bucket’, ‘aws_s3_bucket_public_access_block’ and ‘aws_s3_bucket_policy’ resource blocks. ‘aws_s3_bucket’ resource block used in this implementation is to create S3 bucket which takes ‘bucket’ attribute to specify name of the bucket. Whereas, remaining two blocks are used to specify the bucket access. ‘aws_s3_bucket_policy’ block contains S3 object policy in JSON format that enable read and write access to the S3 bucket.

5.3 Proposed Approach Implementation

5.3.1 Traditional Execution

Traditional approach will execute terraform separately by using multiple Jenkins pipeline. In this approach terraform scripts are stored in different repository and they are executed with the help of different Jenkins pipeline. The execution of those scripts are automated by inter-linking Jenkins pipeline that execute next pipeline after completion of previous one. Each pipeline executes terraform init and apply command on each terraform scripts. This approach does not include python programming language for automating the execution process.

5.3.2 Sequential Execution

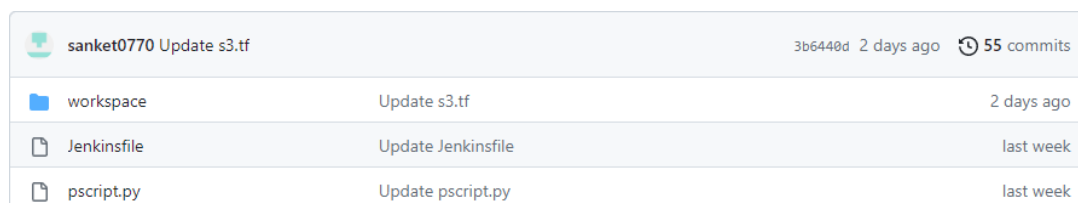
Sequential approach for implementing the AWS resource with the help of terraform is same as parallel execution approach. This approach also incorporates python 3.8 to automate resource deployment process but this approach does not implement Thread-Level Parallelism to ensure concurrency in execution. In this approach also, Jenkins pipeline is used to trigger the python script. After triggering, python script will fetch the terraform script but it will execute this scripts one after other that implement sequential execution of terraform scripts.

5.3.3 Parallel Execution

The implementation of parallel execution approach includes python programming language that enables Thread-Level Parallelism (TLP) in the resource provisioning process. Python script defined in this approach triggered remotely with the help of Jenkins pipeline. After triggering, python script retrieves the terraform scripts from GitHub and assigned it to the individual thread for concurrent execution. Each thread in the execution process will execute the terraform init command on all terraform scripts to initialize and after completion it will start execution of terraform apply command to implement changes.

5.4 Implementation of GitHub and Jenkins

In this project, GitHub and Jenkins also has been used. GitHub is used as a source control repository that stored python script that enable parallel execution and workspace directory that contain terraform script used to provision AWS resources. Whereas, Jenkins used as a deployment server that fetch terraform script and python code form GitHub repository and provision the resources as per configuration.



sanket0770 Update s3.tf		3b6448d 2 days ago 55 commits
workspace	Update s3.tf	2 days ago
Jenkinsfile	Update Jenkinsfile	last week
pscript.py	Update pscript.py	last week

Figure 5: Jenkins stored in GitHub

As shown in the Figure 5, Jenkins file is created in a same GitHub repository that contains terraform scripts. Whenever a pipeline built in Jenkins is executed, the python command stored in the Jenkins file will also get executed.

On complete execution, terraform scripts stored in the GitHub repositories will be executed. This execution will create AWS resources that are specified in configuration such as AWS RDS, Elastic beanstalk and S3 as shown in Figure 6.

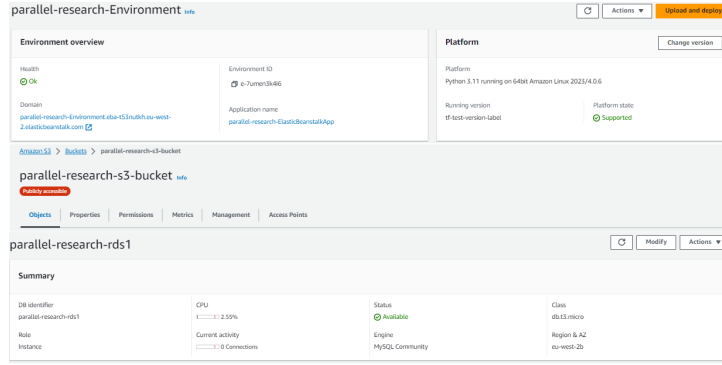


Figure 6: Deployed AWS resources

5.5 Web Application Implementation

For the demonstration purpose, as shown in Figure 7, this project contains one web application that utilize AWS resources which was created during the deployment process. This web application is created using Django framework of python programming language and will be deployed on AWS elastic beanstalk environment which was created during execution of parallel code. Also, this web application incorporates AWS S3 bucket and RDS which was created during execution that allow web application to store Data and images uploaded by the user.

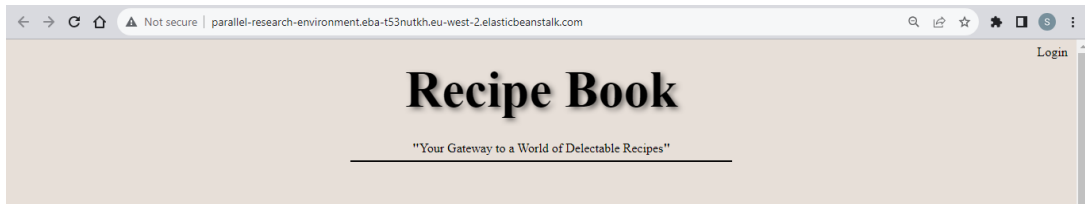


Figure 7: Django-based Web Application

6 Evaluation

The evaluation of the research is critical in determining the efficacy of the suggested methodology for optimizing AWS resource provisioning deployment time using thread-level parallelism. This section provides a thorough examination of the outcomes received from three test cases. The performance analysis of proposed research work was conducted based on three different approach for executing terraform script. First approach performs parallel execution of the terraform script. Second approach performs sequential execution that is non-parallel execution of the terraform scripts. In both approaches, python script is used for executing terraform scripts. Whereas, In the third approach, the terraform scripts were running one after the other utilizing several Jenkins pipelines rather than python scripts. In this approach, total execution time is measured as the sum of all Jenkins pipeline execution times used during the deployment.

The evaluation of each approach primarily focuses on the time taken by the approaches for deploying the AWS resources. Furthermore, it also evaluates additional metrics such

as CPU utilization and network utilization. In all three approaches same AWS resources were provisioned using GitHub, Jenkins and terraform. AWS resources provisioned in this approaches are AWS relational Database Service (RDS), AWS Elastic Beanstalk and AWS S3.

6.1 Experiment 1: Traditional Execution

In traditional execution, instead of using python script, multiple Jenkins pipelines were configured for executing the terraform scripts. These pipelines were set up in such a way that it would start the following pipeline after the previous one finished successfully.

6.1.1 Execution Time

In this approach, multiple Jenkins pipelines were executed for deploying AWS RDS, Elastic Beanstalk and AWS S3, hence the total execution time for the deployment of AWS resources was calculated as the sum of all pipeline execution times. After provisioning all AWS resources with GitHub and Terraform, as illustrated in Figure 8 , it was discovered that total execution takes around 7 minutes and 23 seconds.

S	W	Name	Last Success 1	Last Failure	Last Duration
✓	☀	OFOexecution	1 day 3 hr #6	N/A	21 sec
✓	☀	OFOexecution_eb	1 day 3 hr #4	N/A	2 min 43 sec
✓	☀	OFOexecution_rds	1 day 3 hr #2	N/A	4 min 19 sec

Figure 8: Pipeline for traditional execution along with deployment time

6.1.2 CPU Utilization

During the execution of multiple Jenkins pipeline for provisioning AWS resources in traditional execution approach, continuity of the spikes was observed in CPU utilization. As shown in Figure 9, during the execution it was observed highest spike with 43.59% CPU utilization.

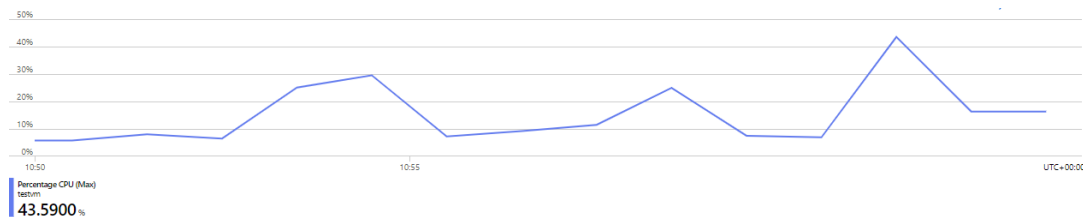


Figure 9: CPU utilization for traditional execution

6.1.3 Network Utilization

In addition to CPU utilization, it detects continual spikes in incoming and outgoing network utilization. The graph shown in Figure 10 demonstrates maximum network utilization, with a maximum of 332.4Kib of incoming data and 802.6Kib of outgoing data.

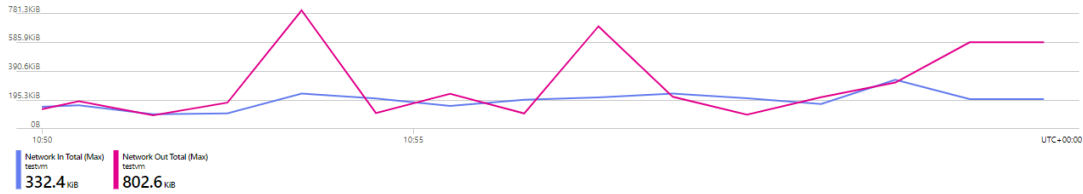


Figure 10: Network utilization for traditional execution

6.2 Experiment 2: Sequential (Non-parallel) Execution

For sequential execution, python code used for execution of the terraform script does not include any module to facilitate thread-level parallelism. In this approach, terraform scripts were executed one after other in sequential manner.

6.2.1 Execution Time

After provisioning AWS resources using terraform scripts and sequential python code with the help of GitHub and Jenkins, as shown in Figure 11, it was observed that it takes around 12 minutes for complete execution of all terraform.

S	W	Name	Last Success	Last Failure	Last Duration
✓	☀	Nonparallel-script	19 min #7	N/A	12 min ▶

Figure 11: Pipeline for sequential execution along with deployment time

6.2.2 CPU Utilization

It was noted that rapid variations in CPU utilization occurred while performing the Jenkins pipeline for deploying AWS resources utilizing sequential python code and Terraform. As shown in Figure 12, the highest spike in CPU utilization was observed, with a maximum of 74.42% CPU utilization.

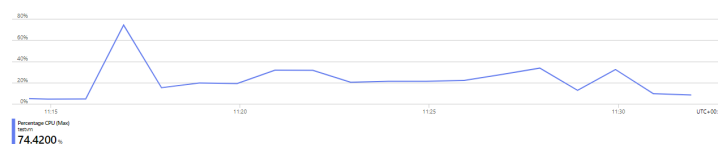


Figure 12: CPU utilization for sequential execution

6.2.3 Network Utilization

It also noticed an immense rise in both incoming and outgoing network traffic throughout the execution, indicating significant data transfer activity. The graph in Figure 13 displays both incoming and outgoing network traffic. The largest rise was noticed in inbound network traffic with 555.9KiB utilization, followed by outbound network traffic with 1.6MiB consumption.

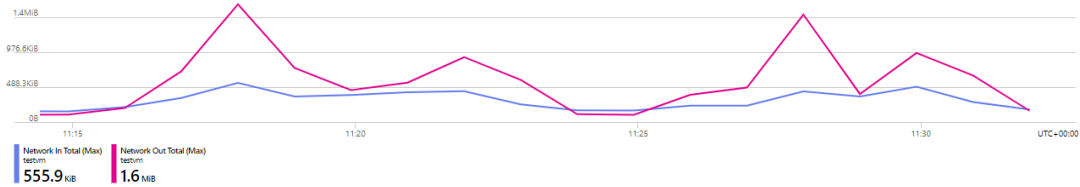


Figure 13: Network utilization for sequential execution

6.3 Experiment 3: Parallel Execution

In parallel execution, python code used for execution of terraform scripts includes concurrent.futures library that provides thread-level parallelism. In this approach, library function assigns terraform execution to the individual thread that carried out execution task in parallel manner.

6.3.1 Execution Time

After provisioning AWS resources using terraform script and parallel python code with the help of GitHub and Jenkins, as shown in Figure 14, it was noticed that it takes approximately 5 minutes and 9 seconds which is faster than sequential as well as traditional execution approach.

S	W	Name 1	Last Success	Last Failure	Last Duration
✓	☀	parallel-script	1 hr 3 min #71	3 days 23 hr #56	5 min 9 sec ▶

Figure 14: Pipeline for parallel execution along with deployment time

6.3.2 CPU Utilization

During the execution of terraform scripts concurrently with the help of Thread-Level Parallelism employing python script, a sudden increase in CPU utilization was noted. as shown in Figure 15, It observed maximum CPU use with a highest spike of 65.44%.

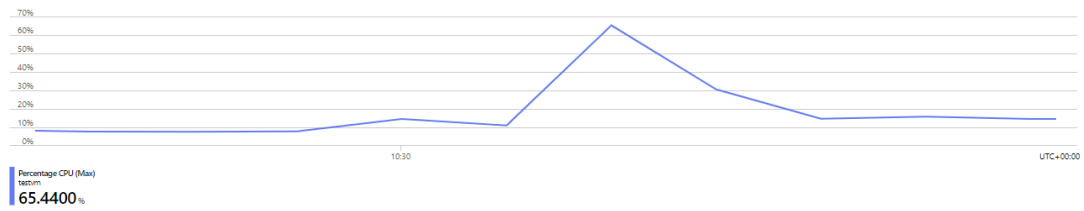


Figure 15: CPU utilization for parallel execution

6.3.3 Network Utilization

Simultaneously, it detected substantial network use activity, suggesting incoming and outgoing data traffic over the network. The graph in Figure 16 shows both incoming and outgoing network traffic, with a maximum of 758.7Kib of incoming data and 1.3Mib of outgoing data

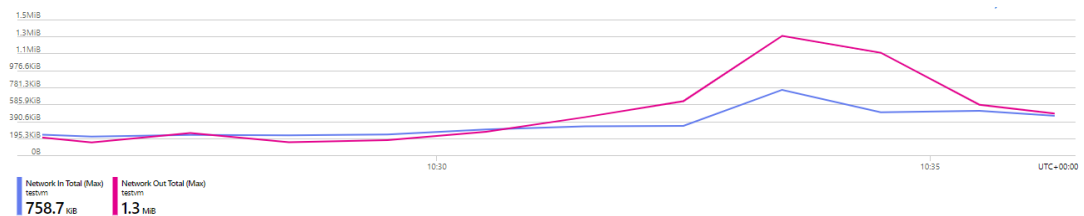


Figure 16: Network utilization for parallel execution

6.4 Discussion

The graph below represents the differences between CPU utilization, Network utilization and execution time taken by each proposed approach to provision AWS resources by running terraform scripts with the help of GitHub and Jenkins.

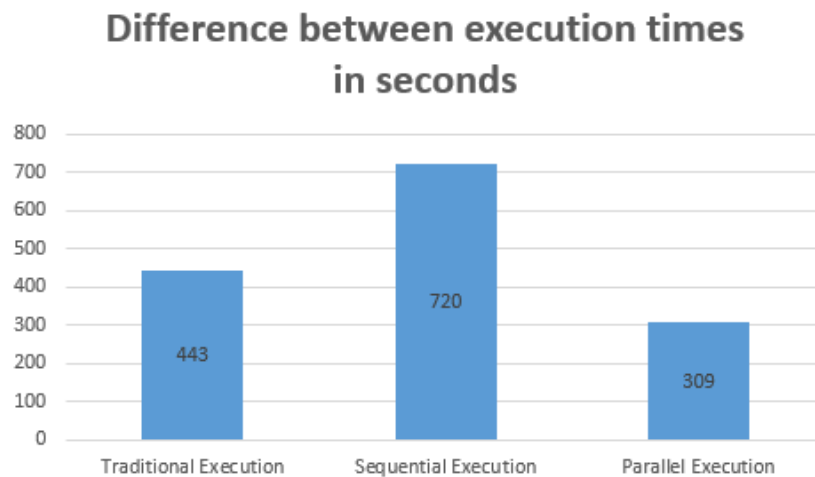


Figure 17: Difference between execution times in seconds

As shown in Figure 17, the traditional execution of terraform scripts took 7 minutes and 23 seconds that is approximately 443 seconds. Whereas, sequential execution took 12 minutes to complete the execution that is equals to 720 seconds and parallel execution took 5 minutes and 9 seconds which is 309 seconds for the execution. After analysing the graph shown in Figure 17 it is noticeable that the parallel execution of terraform scripts for provisioning AWS resources such as AWS RDS, Elastic Beanstalk, and AWS S3 described in this study consumed less time than the traditional and sequential execution approaches

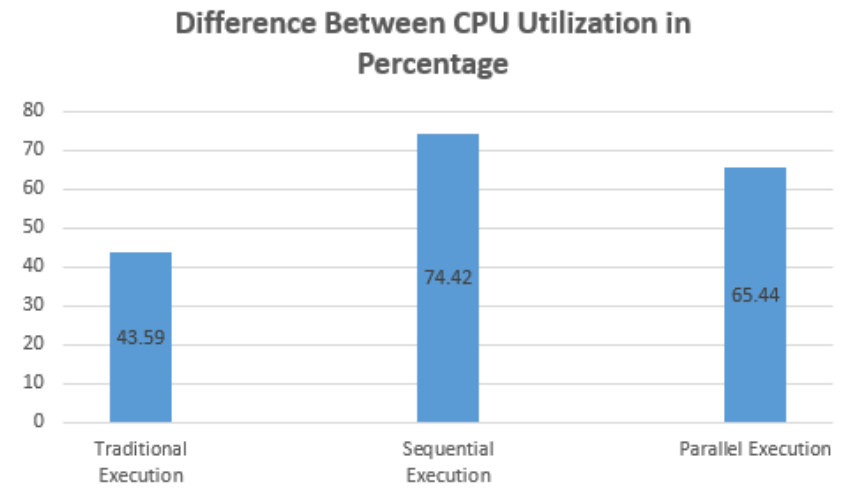


Figure 18: Difference between CPU utilization in percentage

In case of CPU utilization, it observed different values while implementing different approach for deploying resources using terraform. as shown in Figure 18, it noticed that, traditional approach utilized 43.59% CPU whereas, sequential approach utilized 74.42% CPU and parallel execution approach utilized 65.44% while executing terraform scripts that is lower than sequential execution and higher than traditional execution.

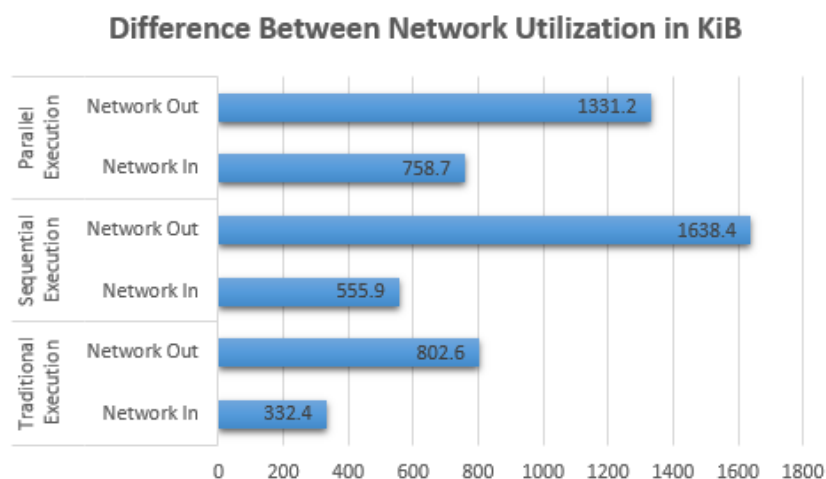


Figure 19: Difference between network utilization in KiB

Furthermore, in terms of network utilization as shown in Figure 19, traditional ex-

ecution consumed less network bandwidth. Whereas, sequential and parallel execution demonstrate equal network utilization while execution of the terraform scripts using GitHub and Jenkins pipeline.

As a result, it concludes that, the methodology proposed in this research work is successful to achieve optimization in the AWS resource deployment using terraform. It is proved that parallel deployment of terraform script takes less execution time as compared to the sequential and traditional execution.

7 Conclusion and Future Work

In the proposed work, the integration of thread-level parallelism with Infrastructure-as-Code (IaC) for AWS resource provisioning presents a promising approach to optimize execution time and enhance the efficiency of the deployment process. The proposed system architecture, utilizing GitHub for source control, Jenkins for automation, and Python 3.8 for implementing thread-level parallelism, demonstrates a structured and scalable solution.

In conclusion, the proposed methodology in this research work showcase significant optimization in AWS resource deployment by integrating terraform with Thread-Level Parallelism (TLP). Through extensive experiment and analysis, the parallel execution proposed in this methodology demonstrates advantages over the sequential and traditional execution of AWS resource deployment.

As the system evolves, future work should focus on optimization, scalability, error handling, monitoring, security, and cross-cloud compatibility. By addressing these aspects, the research can contribute to the development of a robust and versatile solution for efficient infrastructure provisioning in cloud environments. The continuous refinement of this approach will be crucial in meeting the evolving demands of modern cloud-based applications and services.

References

- Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M. and Tamburri, D. A. (2017). Devops: Introducing infrastructure-as-code, *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 497–498.
- Aziz, Z. A., Abdulqader, D. N., Sallow, A. B. and Omer, H. K. (2021). Python parallel processing and multiprocessing: A rivew, *Academic Journal of Nawroz University* **10**(3): 345–354.
- Babuji, Y., Brizius, A., Chard, K., Foster, I., Katz, D. S., Wilde, M. and Wozniak, J. (2017). Introducing parsl: a python parallel scripting library, *Zenodo2017*.
- Brikman, Y. (2022). *Terraform: Up and Running*, " O'Reilly Media, Inc."
- Chansup Byun, William Arcand, D. B. B. B. V. G. M. H. M. H. H. J. M. J. K. K. A. K. P. M. L. M. G. M. J. M. A. P. A. R. A. R. S. S. C. Y. J. K. (2022). ppython: Implementing partitioned global array semantics in python for parallel programming, *Massachusetts Institute of Technology*.

- de Carvalho, L. R. and Patricia Favacho de Araujo, A. (2020). Performance comparison of terraform and cloudify as multicloud orchestrators, *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 380–389.
- Galvez, J. J., Senthil, K. and Kale, L. (2018). Charmpy: A python parallel programming model, *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 423–433.
- Guerriero, M., Garriga, M., Tamburri, D. A. and Palomba, F. (2019). Adoption, support, and challenges of infrastructure-as-code: Insights from industry, *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 580–589.
- Ikeshita, K., Ishikawa, F. and Honiden, S. (2017). Test suite reduction in idempotence testing of infrastructure as code, *Tests and Proofs: 11th International Conference, TAP 2017, Held as Part of STAF 2017, Marburg, Germany, July 19–20, 2017, Proceedings 11*, Springer, pp. 98–115.
- Ivanova, D., Borovska, P. and Zahov, S. (2018). Development of paas using aws and terraform for medical imaging analytics, *AIP Conference Proceedings*, Vol. 2048, AIP Publishing.
- Kim, T., Cha, Y., Shin, B. and Cha, B. (2021). Survey and performance test of python-based libraries for parallel processing, *The 9th International Conference on Smart Media and Applications, SMA 2020*, Association for Computing Machinery, New York, NY, USA, p. 154–157.
URL: <https://doi.org/10.1145/3426020.3426057>
- Morris, K. (2016). *Infrastructure as code: managing servers in the cloud*, ” O’Reilly Media, Inc.”.