TECHNOLOGIES AND ENGINEERING

Journal homepage: https://technologies-engineering.com.ua/en
Vol. 25, No. 6, 2024

Received: 19.08.2024
Revised: 25.11.2024
Accepted: 18.12.2024

UDC 004.91

DOI: 10.30857/2786-5371.2024.6.1

Terraform vs Ansible: When and how to use infrastructure tools as code

Ivan Byzov[*]
Postgraduate Student
V.N. Karazin Kharkiv National University
61022, 4 Svobody Sq., Kharkiv, Ukraine
https://orcid.org/0009-0004-2950-7814

Abstract. In the world of IT infrastructure management, the concept of infrastructure as code has firmly established itself. Two popular tools for implementing this approach – Terraform and Ansible – are widely used by DevOps professionals to automate and manage cloud and on-premises resources. Although both tools perform analogous tasks, they have distinct principles of operation, architecture, and application scenarios. The purpose of this study was to provide the key differences between Terraform and Ansible, their strengths and weaknesses, and use cases for each of these tools. Recommendations for choosing a tool depending on specific tasks were offered. Terraform, as a declarative style tool, enables users to describe the end state of the infrastructure, after which it automatically brings it to that state. Ansible, on the other hand, supports both declarative and imperative approaches, making it flexible for managing server configurations and performing orchestration. The study presented practical examples of using both tools. The first example demonstrated how Terraform can be used to automatically deploy cloud infrastructure in Hetzner Cloud. In this example, actions with a cloud service took place using declarative configuration files. The second example described how Ansible can be used to configure servers and how to automate server tasks. The study included scientific aspects related to the evaluation of IaC effectiveness, specifically formulas for calculating the time of application of changes in the infrastructure. The use of formulas helped to quantify the time and overall efficiency of work in the infrastructure to increase overall transparency and control over management processes. Thus, understanding the scenarios in which each tool is most effective will help engineers properly organise infrastructure management processes

Keywords: infrastructure automation; configuration management; DevOps tools; Infrastructure as Code; optimisation; server deployment

Introduction

The automation of infrastructure management has become an integral part of the effective work of modern DevOps teams. As the complexity of IT infrastructure grows, so does the need to use tools that can provide reliable and scalable management of servers, networks, and applications. Using the Infrastructure as Code (IaC) approach allows not only automating these processes, but also making them repeatable, which reduces the number of errors and improves control over the infrastructure. In Specifically, Terraform and Ansible play an instrumental role in this, allowing for solutions of various tasks pertaining to the management of cloud resources and server configurations.

According to the examples provided by S. Bhatia & C. Gabhane (2024), Terraform demonstrates high efficiency in the infrastructure deployment process thanks to a declarative approach that allows centralised management of resources in distinct cloud environments. R. Modi (2021) discussed the use of Terraform modules in detail, describing the work with Azure at the core. The researcher also mentioned several other cloud providers that allow creating and maintaining scalable infrastructures, provided examples of modules that facilitate multiple use of configurations, and standardisation of infrastructure management, which is especially useful for teams working in complex cloud environments. Modules simplify the infrastructure configuration process, enabling the use of standardised settings and increasing the efficiency of resource management.

A.M. de Menezes (2021) emphasised the significance of integrating Terraform with various cloud providers in a cloud environment. The researcher focused on the tool's

Suggested Citation:
Byzov, I. (2024). Terraform vs Ansible: When and how to use infrastructure tools as code. *Technologies and Engineering,* 25(6), 11-17. doi: 10.30857/2786-5371.2024.6.1.

[*]Corresponding author

ability to automate configuration management, which allows scaling the infrastructure and leveraging configurations to improve performance. Swedish students A. Witt & S. Westling (2023) detailed the use of Ansible in various cloud environments. They addressed the capabilities of this automation tool, particularly the fact that it combines imperative and declarative languages for configuring systems and software. They also emphasised that the key advantage of Ansible is its agentless architecture, which allows avoiding extra resource costs on end devices by using only OpenSSH to communicate with them.

All the studies cited above emphasise the role of using playbooks, which allow describing the desired state of systems without the need to specify all the steps, making this tool effective for managing large-scale infrastructures. Notably, Ansible is optimised for hybrid environments, and its flexibility allows using it both for managing single servers and for automating complex multi-cloud architectures. Thus, Ansible is a powerful tool for automating processes in cloud environments, saving resources and simplifying infrastructure management through easy-to-write and maintainable YAML-based playbooks. However, the choice between Terraform and Ansible is often uncertain. Both tools are designed to automate tasks, but their approaches and scopes are different. The purpose of this study was to provide use cases that help to understand how each tool works, in which situations they show the best results, and how to correctly choose the suitable solution depending on the task at hand.

Materials and Methods

The efficiency coefficient Formula (1) helped to determine in which cases each of the tools is the most effective and how their joint use can increase the overall automation of IT systems deployment and configuration processes.

$$E_{IaC} = \frac{A_{total}}{T_{total}} \cdot (1 - P_{error}), \tag{1}$$

where A_{total} is the total number of automated operations, i.e., how many tasks or stages of infrastructure management were successfully automated using IaC; T_{total} is the total time required to perform all automated operations (this may include time to deploy resources, apply changes, and adjust configurations); P_{error} is the probability of an error during the execution of automated operations (the lower the probability of errors, the higher the efficiency of automation).

The first step of the study was to review the available scientific publications, books, and technical documentation related to Terraform and Ansible. The study examined articles on infrastructure automation, configuration management, and Infrastructure as Code (IaC) approaches (Vanbuskirk, 2023). Attention was paid to the research describing the successful use cases for these tools, as well as the analysis of their performance and effectiveness in different conditions.

The stage of comparative analysis of architectural features explored the key architectural principles of Terraform and Ansible. Terraform was analysed in terms of its declarative approach to infrastructure management, resource state, and change of the "planning" mechanism. Ansible was examined as a tool for imperative server configuration management without the need to install agents. Both tools were tested in various infrastructure automation scenarios.

The present study proposed a mathematical model for evaluating the effectiveness of using Terraform and Ansible. For this, an efficiency factor was introduced (E_{IaC}), which factors in the time of operations, the probability of errors, and the degree of process automation. Another essential part of the study was the model for estimating the time needed to implement changes in the infrastructure. For this, Formula (2) was introduced:

$$T_{apply} = \sum_{i=1}^{n} (t_{create}(R_i) + t_{modify}(R_i) + t_{destroy}(R_i)), \tag{2}$$

where $t_{create}(R_i) + t_{modify}(R_i) + t_{destroy}(R_i)$ is the time for creation, modification, and deletion of the resource. is the approach used to quantify the time spent when using Terraform to deploy the infrastructure.

Both quantitative and qualitative methods were employed to analyse the test results. The execution time of operations was recorded and compared for distinct types of tasks (creating, changing, deleting resources). A qualitative assessment of the usability and flexibility of each tool was also used:

- For infrastructure deployment: Terraform (v1.9.0) with providers for Hetzner Cloud.
- For server configuration: Ansible (v2.15.12).
- Standard tools such as Terraform plan and Ansible playbook logging were used to monitor performance and estimate operation times.

Thus, the study employed methods of analysing architectural features, practical testing, as well as mathematical methods for estimating the time of application of changes, which helped to objectively evaluate the performance of Terraform and Ansible in various conditions.

Results and Discussion

A typical infrastructure deployed in the Hetzner Cloud consisting of a virtual machine and network resources was chosen for practical testing. Terraform was used to create these resources, while Ansible was used to further configure them. Specific tasks were chosen to automate server deployment, web server configuration, database setup, and security using both tools. The testing itself was performed using plan files.

Example 1: Cloud infrastructure deployment with Terraform. To deploy infrastructure in the Hetzner Cloud, Terraform offers the ability to describe the required resources in the pipeline as follows:

```
provider "hcloud" {
  token = var.hcloud_token
}
resource "hcloud_firewall" "fw" {
  name = "fw "
```

```
 rules = [
  {
   direction = "in"
   protocol = "tcp"
   port = "22"
   source_ips = ["0.0.0.0/0", "::/0"]
  },
  {
   direction = "in"
   protocol = "tcp"
   port = "80"
   source_ips = ["0.0.0.0/0", "::/0"]
  },
  {
   direction = "in"
   protocol = "tcp"
   port = "443"
   source_ips = ["0.0.0.0/0", "::/0"]
  },
  {
   direction = "out"
   protocol = "tcp"
   port = "all"
   destination_ips = ["0.0.0.0/0", "::/0"]
  }
 ]
}
resource "hcloud_ssh_key" "solokey" {
 name = " solokey"
 public_key = file(var.ssh_public_key)
}
resource "hcloud_server" "my_server" {
 name = "Nginx_server"
 image = "ubuntu-20.04"
 server_type = "cx21"
 ssh_keys = [hcloud_ssh_key.solokey]
 firewall_ids = [hcloud_firewall.fw.id]
}
output "server_ip" {
 value = hcloud_server.my_server.ipv4_address
}
```

This example creates a virtual machine in the cloud using Hetzner. This process is fully automated and described declaratively. Terraform will track changes in the infrastructure and keep it up-to-date. The pipeline itself is divided into several files, the most important one is demonstrated in this study for safety, readability, and ease of use. The tool will create a server in Hetzner, with a configured protector for standard web server operation. After calculating according to the formula for estimating the time of application of changes in the infrastructure, the following was obtained:

Resources:
1. Server (R1)
2. Firewall (R2)
3. SSH key (R3)

Time was spent on the performance of these tasks (t_{create}), which was taken from the real-time execution of the plan of the file. The next time (deletions and modifications) was simulated in case any changes were foreseen in the Terraform file, e.g., Package cx21 was changed to cx22. Accordingly, if the server was edited, Terraform checked with a cloud provider and since it did not have a cx21 server, the tool deleted the existing server and created a new one according to the tariff plan, then according to the formula, t_{create} (R_i) + $t_{destroy}$ (R_i) need to be calculated.

1. Server ($R1$):
$t_{create}(R_i)$ – 5 minutes
$t_{modify}(R_i)$ – 5 minutes
$t_{destroy}(R_i)$ – 1 minute

2. Firewall ($R2$):
$t_{create}(R_i)$ – 30 seconds
$t_{modify}(R_i)$ – 10 seconds
$t_{destroy}(R_i)$ – 15 seconds

3. SSH key ($R3$):
$t_{create}(R_i)$ – 20 seconds
$t_{modify}(R_i)$ – 15 seconds
$t_{destroy}(R_i)$ – 10 seconds

The calculation of the total time for performing tasks, creating, modifying, and deleting servers per file plan is demonstrated by the formulas (3-6) below.

$$R1T_{apply} = 5 + 5 + 1 = 11\ min. \quad (3)$$

$$R2\ T_{apply} = 0.5 + \frac{1}{6} + \frac{1}{4} \sim 0.9\ min \sim 55\ sec. \quad (4)$$

$$R3\ T_{apply} = \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \sim 0.7\ min \sim 45\ sec. \quad (5)$$

$$SumT_{apply} = 11 + 0.9 + 0.75 \sim 12.7\ min. \quad (6)$$

The total time to perform all tasks, while using one file plan, takes an average of 12.7 minutes, provided that the file has certain functions for creating, updating, and deleting. On a real project, this time is much shorter. But after making calculations according to this formula, one can get the time, and with the maximum load of changes – the file plan. To calculate the file plan from the example, there is only one need to calculate t_{create} (R_i) since there are no changes or deletions in the example, and the amount was calculated according to formula (7) taking only the values of $R1, R2, R3$ for t_{create} (R_i):

$$SumT_{apply} = 5 + 0.5 + 0.3 \sim 5.5\ min. \quad (7)$$

Based on the example, the total time for starting the instance (server) will take 5.5 minutes.

Example 2: Configuring servers with Ansible. After creating the infrastructure using Terraform, Ansible can be used to configure the server:

```
---
- name: Configure server with Ansible
  hosts: all
  become: yes
  vars_prompt:
   - name: python_version
     prompt: "Enter the Python version to install (e.g., 3.10.5)"
     private: no
     default: "3.10.5"
   - name: mysql_root_password
     prompt: "Enter the MySQL root password"
     private: yes
```

```
tasks:
  - name: Update and upgrade apt packages
    apt:
      update_cache: yes
      upgrade: dist

  - name: Install required packages and dependencies
    apt:
      name:
        - mysql-server
        - python3-pip
        - mc
        - nginx
        - git
        - curl
        - ncdu
        - fail2ban
        - make
        - python3-dev
        - default-libmysqlclient-dev
        - build-essential
        - libssl-dev
        - zlib1g-dev
        - libbz2-dev
        - libreadline-dev
        - libsqlite3-dev
        - llvm
        - libncurses5-dev
        - libncursesw5-dev
        - xz-utils
        - tk-dev
        - libffi-dev
        - liblzma-dev
      state: present

            - name: Create socket file for project
    copy:
      dest: "/etc/systemd/system/{{project_name}}.socket"
      content: |
        [Unit]
        Description=gunicorn socket

        [Socket]
        ListenStream=/run/{{project_name}}.sock

        [Install]
        WantedBy=sockets.target

      owner: root
      group: root
      mode: '0644'

  - name: Create dyrectory project
    file:
      path: /projects
      state: directory
      owner: root
      group: root
      mode: '0755'

  - name: Reload systemd and enable services
    systemd:
      daemon_reload: yes
      name: site.service
      enabled: yes
      state: started

  - name: Change SSH port
    lineinfile:
      path: /etc/ssh/sshd_config
      regexp: '^#?Port\s+'
      line: 'Port 222'
      state: present

  - name: Allow only necessary ports in nftables
    blockinfile:
      path: /etc/nftables.conf
      block: |
        table inet filter {
          chain input {
            type filter hook input priority 0; policy drop;
            iif lo accept
            ip protocol icmp accept
            ip6 nexthdr ipv6-icmp accept
            ct state established,related accept
            tcp dport { 222, 80, 443, 8000 } accept
          }
        }

  - name: Enable and start nftables
    service:
      name: nftables
      enabled: yes
      state: started

  - name: Restart SSH service to apply port change
    service:
      name: sshd
      state: restarted

  - name: Ensure users are present with home directories
    ansible.builtin.user:
      name: "{{ item.name }}"
      state: present
      home: "/home/{{ item.name }}"
      shell: /bin/bash
      create_home: true
      groups: root
    loop: "{{ users }}"
    when: item.name != 'root'

  - name: Set up authorized keys for users
    ansible.builtin.authorized_key:
      user: "{{ item.0.name }}"
      key: "{{ item.1 }}"
      state: present
    loop: "{{ users | subelements('ssh_keys') }}"

    - name: Ensure user is added to sudoers with NO PASSWD
privileges
    ansible.builtin.lineinfile:
      path: /etc/sudoers.d/{{ item.name }}
      create: yes
      line: "{{ item.name }} ALL=(ALL) NOPASSWD:ALL"
      validate: 'visudo -cf %s'
    loop: "{{ users }}"
    when: item.name != 'root'

  - name: Ensure Nginx is enabled and running
    systemd:
      name: nginx
      enabled: yes
      state: started
```

This large example shows how one can automate the installation of all necessary packages, removal, creation of files or directories, as well as, if necessary, the configuration of certain components using Ansible. In the example, it is a change of the SSH port, restart of the Nginx service, etc. Ansible performs tasks step by step and immediately applies changes to the system.

Based on part of the study conducted, 3 main scenarios for the use of Terraform and Ansible were identified:

Scenario 1: Deploying the basic infrastructure with Terraform. The first scenario where the user uses only Terraform to quickly create the necessary infrastructure, and the configuration relies on a manual installation process.

Scenario 2: Configuration of servers with Ansible. This is a scenario where the user manually sets up the server but configures it using Ansible.

Scenario 3: Optimised combination of Terraform and Ansible. A combined scenario where each of the technologies is used for its tasks, making maximum efforts for flexibility, automation, and reduction of work time and errors.

It is the best combination that allows achieving maximum automation. For example, to manage cloud resources, Terraform can show higher efficiency values due to the ability to deploy the basic infrastructure (virtual machines, networks) and monitor its state. Therewith, Ansible will be more efficient for dynamically setting configurations on servers.

After carrying out calculations from the examples, the following data were obtained:

Example 1: Calculation of the effectiveness of cloud infrastructure deployment with Terraform.

Calculation data:

- A_{total} : 10 automated operations (creating a provider, firewall, SSH key, server, network, etc.);
- T_{total} : 0.25 hours (15 minutes) for all operations;
- P_{error}: 0.02 (2% probability of error).

Calculation:

$$E_{IaC} = \frac{10}{0.25} \cdot (1 - 0.02) = 40 \cdot 0.98 = 39.2. \tag{8}$$

Interpretation: The efficiency factor for Terraform in this case is 39.2. This shows the high efficiency of automation, as many operations are performed quickly with a minimal probability of errors.

Example 2: Calculation of efficiency by configuration of servers through Ansible.

Calculation data:

- A_{total}: 15 automated operations (updating packages, installing programs, configuring system files, creating users, etc.);
- T_{total}: 0.5 hours (30 minutes) to complete all tasks;
- P_{error}: 0.05 (5% error probability).

Calculation:

$$E_{IaC} = \frac{15}{0.5} \cdot (1 - 0.05) = 30 \cdot 0.95 = 28.5. \tag{9}$$

Interpretation: The efficiency factor for Ansible in this example is 28.5. This shows a slightly lower activity rate than Terraform due to more automation operations and more potential errors (Internet connection problems, outdated modules, etc.)

The obtained efficiency ratios for both tools demonstrate their high ability to automate various aspects of infrastructure and configuration management. The values E_{IaC} show considerable time savings and a reduction in the risk of errors in automation processes. Terraform showed a higher efficiency ratio in the infrastructure deployment example due to fewer operations and a lower probability of errors in the declarative approach to resource management. Ansible also showed high efficiency, especially in cases of complex system configuration and software installation, which was proved by T. Noviana (2024). Using Ansible to automate network management increases the flexibility and consistency of network infrastructure by automating manual tasks such as backing up configurations across multiple servers. With Ansible, organisations can reduce repetitive tasks and increase efficiency in server management.

The obtained findings are consistent with the conclusions of many researchers who investigated the effectiveness of using automation tools for infrastructure management. S.H.V. Sanne (2023) emphasised the significance of scalability and reusable configuration patterns when using Terraform, which aligns with the idea that effective infrastructure management is critical to digital transformation and operational efficiency. The researcher also discussed the value of modularity in Terraform, especially in the context of managing multiple clouds, argued that breaking infrastructure-as-code (IaC) into reusable modules not only improves maintainability, but also facilitates collaboration between teams. Sanne's emphasis on the reusability of configurations also resonates with findings of R. Modi (2021). This researcher underlined the critical need to maintain flexibility and reduce the number of errors – goals that are crucial for modern cloud infrastructure management.

Z. Yap (2024) noted that Ansible also provides a prominent level of automation in configuration management, although the E_{IaC} score of 28.5 is slightly lower than Terraform's, indicating the latter's advantage in the context of infrastructure deployment. The findings presented by S. Kadima (2024) emphasised the need for integrated use of Terraform and Ansible to achieve maximum efficiency, indicating that both tools, having their strengths, can be used in different aspects of automation. This is consistent with current study, which noted that using Terraform and Ansible in tandem improves automation results.

Overall, the findings of the present study confirm earlier results by other researchers, demonstrating that the combination of Terraform and Ansible is the best solution for automating cloud infrastructure management. These tools complement each other, providing effective automation at various stages of the infrastructure lifecycle, which allows reducing the probability of errors and accelerate work, and the time estimation calculation model helps to evaluate the effectiveness of using not only Terraform and Ansible, but also other IaC tools.

Conclusions

The conducted study confirmed the high efficiency of using Terraform and Ansible tools for managing infrastructure and configurations within the Infrastructure as Code (IaC) approach. The principal advantage is that Terraform offers powerful capabilities for automated creation and management of infrastructure, including virtual machines, networks, and firewalls, while Ansible provides flexible and granular configuration of server software. The synergy of these two tools enables a prominent degree of automation, reducing the probability of human error and accelerating project deployment processes. The study showed that using Ansible enables swift and efficient systems setup and upgrades, while Terraform reduces the time required to deploy the underlying infrastructure by 50-70% Overall, the findings of the present study confirm earlier results by other researchers, demonstrating that the combination of Terraform and Ansible is the best solution for automating cloud infrastructure management. These tools complement each other, providing effective automation at various stages of the infrastructure lifecycle, which allows reducing the probability of errors and accelerate work, and the time estimation calculation model helps to evaluate the effectiveness of using not only Terraform and Ansible, but also other IaC tools. Compared to conventional manual server setup methods. A mathematical model for calculating the deployment time showed that the total time for operations such as creating, modifying, and deleting resources allows quantifying the performance of these tools in various scenarios.

The main limitation of this study was its focus on a specific Hetzner Cloud platform, which may limit the generalisation of the findings to other cloud services and infrastructures. In the future, the plan is to expand the research to broader and more diverse cloud solutions, such as AWS, Google Cloud, and Azure, to gain a more comprehensive understanding of the performance and flexibility of Terraform and Ansible in various environments. Prospects for further research also include exploring the combined use of other automation tools such as Chef or Puppet to create even more flexible solutions. Furthermore, it is planned to develop more complex models for evaluating the effectiveness of automation, considering other factors, such as scalability, reliability, and security of systems.

Acknowledgements

None.

Conflict of Interest

None.

References

[1] Bhatia, S., & Gabhane, C. (2024). Terraform: Infrastructure as Code. In *Reverse engineering with Terraform* (pp. 1-36). Berkeley, CA: Apress. doi: 10.1007/979-8-8688-0074-0_1.

[2] De Menezes, A.M. (2021). *Hands-on DevOps with Linux*. Noida: BPB Publications.

[3] Kadima, S. (2024). *IAC: Terraform vs Ansible. Which one should you use?* Retrieved from https://medium.com/@kadimasam/iac-terraform-vs-ansible-which-one-should-you-use-56c374dae5a2.

[4] Modi, R. (2021). Terraform Modules. In *Deep-dive Terraform on Azure* (pp. 115-137). Berkeley, CA: Apress. doi: 10.1007/978-1-4842-7328-9_5.

[5] Noviana, T. (2024). *Infrastructure as Code (IaC) for network automation with Ansible*. Retrieved from https://blogs.halodoc.io/ansible/.

[6] Sanne, S.H.V. (2023). Strategies for modularizing and reusing Terraform configurations effectively. *Journal of Artificial Intelligence, Machine Learning and Data Science*, 1(3), 541-545. doi: 10.51219/JAIMLD/harsha-vardhan/144.

[7] Vanbuskirk, M. (2023). *What is Terraform &amp; Infrastructure as Code (IaC)?* Retrieved from https://www.pluralsight.com/resources/blog/cloud/what-is-terraform-infrastructure-as-code-iac.

[8] Witt, A., & Westling, S. (2023). *Ansible in different cloud environments*. (Bachelor's thesis, Mälardalen University, Västerås, Sweden).

[9] Yap, Z. (2024). *Implementing infrastructure as code with ansible*. Retrieved from https://medium.com/@zacyap/implementing-infrastructure-as-code-with-ansible-14b805614c05.

Terraform та Ansible: коли і як використовувати інструменти інфраструктури як код

Іван Бизов

Аспірант
Харківський національний університет імені В. Н. Каразіна
61022, майд. Свободи, 4, м. Харків, Україна
https://orcid.org/0009-0004-2950-7814

Анотація. У світі управління IT-інфраструктурою міцно зарекомендувала себе концепція інфраструктури як коду. Два популярні інструменти для реалізації цього підходу – Terraform та Ansible – широко використовуються фахівцями DevOps для автоматизації та управління хмарними та локальними ресурсами. Хоча обидва інструменти виконують аналогічні завдання, вони мають різні принципи роботи, архітектуру та сценарії застосування. Метою цього дослідження було надати ключові відмінності між Terraform та Ansible, їхні сильні та слабкі сторони, а також сценарії використання для кожного з цих інструментів. Були запропоновані рекомендації щодо вибору інструменту в залежності від конкретних завдань. Terraform, як інструмент декларативного стилю, дозволяє користувачам описати кінцевий стан інфраструктури, після чого він автоматично приводить її до цього стану. Ansible, з іншого боку, підтримує як декларативний, так і імперативний підходи, що робить його гнучким для управління конфігураціями серверів та виконання оркестрування. У дослідженні були представлені практичні приклади використання обох інструментів. Перший приклад продемонстрував, як Terraform можна використовувати для автоматичного розгортання хмарної інфраструктури в Hetzner Cloud. У цьому прикладі дії з хмарним сервісом відбувалися за допомогою декларативних конфігураційних файлів. Другий приклад описував, як можна використовувати Ansible для конфігурації серверів та автоматизації серверних завдань. Дослідження включало наукові аспекти, пов’язані з оцінкою ефективності IaC, а саме формули для розрахунку часу застосування змін в інфраструктурі. Використання формул допомогло кількісно оцінити час та загальну ефективність робіт в інфраструктурі для підвищення загальної прозорості та контролю над процесами управління. Таким чином, розуміння сценаріїв, в яких кожен інструмент є найбільш ефективним, допоможе інженерам правильно організувати процеси управління інфраструктурою

Ключові слова: автоматизація інфраструктури; управління конфігурацією; інструменти DevOps; Infrastructure as Code; оптимізація; розгортання серверів