RESEARCH ARTICLE OPEN ACCESS

Optimizing Infrastructure Management using Ansible

Aditi, Vanshika, Alok Kumar
Adobe Inc, Noida, India

ABSTRACT
In the modern landscape of information technology, automation has become a cornerstone for managing complex infrastructures, enhancing operational efficiency, and ensuring consistency across large-scale systems. One of the most widely adopted tools in the domain of IT automation and configuration management is Ansible. Ansible is an open-source automation platform that facilitates tasks such as configuration management, application deployment, intra-service orchestration, and provisioning. Ansible distinguishes itself from other automation tools through its agentless architecture, relying on SSH and Python, and its use of simple, human-readable YAML syntax for playbooks. This ease of use, combined with its powerful capabilities, has made Ansible a popular choice among system administrators, DevOps engineers, and organizations seeking scalable and reliable automation solutions. This paper explores the core architecture of Ansible, issues that arise with the manual setup, how they are overcome by Ansible and advantages of using it. We also compare the time taken for the machine setup of clusters of varying sizes of machines done manually and done using Ansible.

Keywords **—** configuration management, Ansible, machine setup, playbooks, automation, release engineering

I. INTRODUCTION

In modern software development ecosystems, particularly within large-scale enterprise environments, there is an ever-growing need to generate production builds at a high frequency. This rapid build cadence is essential to deliver new features, address software defects, and apply stability improvements with minimum latency, thereby ensuring continuous value delivery to end users [1]. The software development lifecycle in such settings is inherently dynamic, with multiple developers concurrently contributing code to Source Code Management (SCM) tool. SCM platforms such as Git facilitate collaboration through features like Pull Requests (PRs), which act as formal proposals for integrating code changes into the primary codebase [2]. Prior to merging, it is a standard industry practice to generate a complete build from each PR to preserve the stability and integrity of the main branch. This is also required to perform the basic testing on the build. In environments comprising large development teams, dozens or even hundreds of PRs may be raised on a daily basis. These PRs necessitates a significant number of builds that must be executed concurrently on dedicated machines—collectively referred to as the build infrastructure. The reliability and performance of this infrastructure are pivotal to maintain the throughput of software delivery pipeline - an automated process that manages the building, testing and deployment of software from development to production. Each build machine within the infrastructure requires a meticulously configured environment—referred to as the machine setup—which includes specific versions of compilers, build tools, signing utilities, packaging tools, and environment variables tailored to the needs of the product. Consistency in machine setup ensures predictable and reproducible build behavior across various machines and over time. The configuration landscape is not static. Tools and dependencies evolve continuously in response to security patches, feature updates, and platform changes. This necessitates frequent updates to the machine configurations, which fall under the purview of the release engineering team. While initial setup and periodic updates can be performed manually, such an approach is inherently inefficient and error-prone, particularly at scale. Manual configuration introduces a risk of human error and inconsistency, which can result in build failures, increased debugging overhead, and delays in the development lifecycle. As infrastructure scales by scaling horizontally – by adding more machines to support increased parallel builds, repeating manual setup across multiple machines quickly becomes unsustainable. To address these challenges, organizations increasingly adopt Configuration Management (CM) tools, which automate the process of machine setup and maintenance. Tools such as Ansible, an open-source, agentless automation engine, which means it does not require any software or agents to be installed on the target systems allow infrastructure to be defined as code. It ensures consistency, repeatability, and auditability of configurations. By leveraging such tools, the release engineering team can standardize environments across all build machines, eliminate manual interventions, and significantly improve operational efficiency. Automation, in this context, refers to the use of configuration management tools to programmatically handle infrastructure setup and software configuration, thereby reducing setup time and human effort. This also facilitates rapid scaling of infrastructure and minimizes configuration drift across environments resulting in a more resilient and productive software delivery pipeline [3].

The landscape of configuration management and orchestration tools has evolved significantly, offering a diverse set of solutions that cater to varying infrastructure paradigms, ranging from traditional on-premises data centers to highly dynamic cloud-native architectures. These tools offer distinct features and capabilities. Orchestration frameworks are systems or tools designed to automate and manage the execution of complex workflows across distributed components. For the purpose of this study, we evaluated four widely recognized CM tools - Ansible, Chef, Puppet, and SaltStack — and, at the same time, two leading orchestration frameworks - Terraform and AWS CloudFormation. While orchestration tools such as Terraform and CloudFormation are highly effective for provisioning and managing cloud infrastructure through declarative infrastructure-as-code (IaC) approaches, their applicability in environments dominated by on-premises systems is often limited. Given that most of our infrastructure is deployed on-premises, and while orchestration solutions such as Terraform and AWS CloudFormation are available, this study focuses exclusively on evaluating configuration management tools. Each of the shortlisted configuration management tools was examined in detail. Each tool exhibits distinct characteristics across various parameters, as discussed below.

Codebase and Licensing: All four tools are open-source projects. While open-source status was not a strict requirement for selection, this commonality is noted as it offers benefits such as source code accessibility and community-driven improvements [4].

Cloud Platform Compatibility: All tools support deployment and management across leading cloud environments making them versatile for cloud infrastructure management [4].

Tool Type and Infrastructure Model: All tools are configuration management tools, designed to automate the setup, management, and maintenance of infrastructure and software configurations. They modify and update existing servers and configurations instead of rebuilding servers from scratch when changes occur, as seen in immutable infrastructure [4].

Language Paradigm: Chef and Ansible follow a procedural programming model, requiring users to specify the exact the sequence of tasks to be executed. In contrast, Puppet and SaltStack use a declarative model, allowing users to define the desired final state while the tool determines the necessary steps to achieve it [4].

Configuration Tool and Language: Chef utilizes ‘Cookbooks’ consisting of ‘Recipes’ to define configurations in Ruby-based domain-specific language (DSL). Puppet similarly uses ‘Recipes’ and ‘Cookbooks’ in its proprietary PuppetDSL. Ansible uses ‘Playbooks’, written in YAML, while SaltStack employs ‘States’ for configuration definitions, also in YAML, supplemented by Python [5].

Architecture and Operational Model: Chef, Puppet, and SaltStack follow a client-server architecture where managed nodes run agents that regularly pull configuration instructions from a central server. In contrast, Ansible uses an agentless, push-based approach, sending configurations directly from the control node to target systems over SSH on demand, simplifying deployment [5].

Ease of Use and Dependencies: Ansible and SaltStack are generally regarded as easier to learn and use because of their simple syntax and minimal dependencies. Ansible depends only on SSH and Python, whereas Chef, Puppet, and SaltStack need agents installed on managed nodes, resulting in moderate dependency levels. While Chef is powerful, it has a steeper learning curve, and Puppet, though somewhat easier, remains relatively complex. Overall, Ansible and SaltStack stand out for their simplicity and agentless architecture [5].

Intended Use Cases: Chef is commonly favored for developer-centric infrastructure automation and workload deployment. Puppet is widely used for system management, code configuration, and reporting. Ansible stands out in orchestration, deployment, and provisioning alongside configuration management. SaltStack integrates system management with orchestration and deployment capabilities [5].

Scalability: In the case of configuration management tools, scalability means they can handle increasing numbers of servers, devices, configurations, or services without significant degradation in speed, responsiveness, or manageability. All four tools demonstrate high scalability and can effectively manage large complex environments comprising thousands of nodes [6].

Manual setup causes many issues, listed in Table I along with their details, including configuration errors, inefficiencies, and human oversight. Manual setup also lead to inconsistencies across different environments, making it difficult to maintain system reliability. Additionally, the lack of automation in the setup process can lead to prolonged troubleshooting and difficulty in scaling the system for larger operations.

TABLE I

ISSUES WITH MANUAL SETUP

Issue	**Details**
Inconsistency Across Systems	Human errors and inconsistent procedures can cause variations in versions, configurations, and missing dependencies.
Lack of Documentation and Reproducibility	Manual processes are poorly documented, making it difficult to replicate the setup on other machines.
Scalability challenges	Manual installation is slow, tedious, error prone, and impractical for large-scale or dynamic environments.
Configuration Drift	Machines setup manually may drift from their intended state over time, resulting in inconsistent behavior.
No Rollback or Version Control	Do not support rollback or versioning, which complicates change tracking and failure recovery.
Security Risks	Manual installations risk using untrusted or outdated tools and frequently neglect crucial validation and security measures
Slower Recovery and Onboarding	System recovery and new developer onboarding take longer, resulting in more downtime and decreased productivity.
Hard to Audit or Monitor	Absence of audit trails complicates troubleshooting as the system state is not clearly tracked.

Table II presents a comparative analysis of Ansible, Chef, Puppet, and SaltStack based on all of the factors discussed in this section.

TABLE II

COMPARISON OF CONFIGURATION MANAGEMENT TOOLS

	Chef	**Puppet**	**Ansible**	**SaltStack**

Code	Open Source	Open Source	Open Source	Open Source
Cloud Support	All	All	All	All
Type	CM	CM	CM	CM
Infrastructure	Mutable	Mutable	Mutable	Mutable
Language	Procedural	Declarative	Procedural	Declarative
Configuration Tool	Cookbook	Recipes/Cookbook	Playbook	States
Architecture	Client/Server	Client/Server	Client only	Client/Server
Configuration Language	DSL (Ruby)	DSL (PuppetDSL)	YAML (Python)	YAML (Python)
Operational Model	Pull	Pull	Push	Pull
Ease of Use	Moderate-High	Moderate	High	High
Dependencies	Medium	Medium	Minimal	Medium
Management Purpose	Developer Based Infrastructure Automation, Automatized workload deployment	System Management, Code Management, Configuration Automation, Reporting	Orchestration, Deployment, Provisioning	System Management, Orchestration, Deployment
Scalability	High	High	High	High

Our comparative analysis determined that all four tools are open-source, exhibit robust support for integration with major cloud platforms such as AWS, Azure, and Google Cloud and are architecturally designed to scale efficiently in large environments. After careful evaluation, Ansible was selected as the most appropriate tool for our use case. Key factors influencing this decision include its agentless architecture, straightforward installation process, user-friendly syntax (YAML-based playbooks), and high ease of use. These attributes collectively contribute to easier adoption, simplified management, and lower operational overhead compared to the alternatives. head from the text.

II. ANSIBLE BASED AUTOMATION: ARCHITECTURE AND APPLICATION

In this section we discuss the high-level components of Ansible. We cover various scenarios related to changes in build infrastructure or machine setup and explain how these situations can be handled using Ansible. Additionally, we describe how Ansible resolves issues commonly encountered in manual setups.

A. High Level Components

Fig. 1 shows the architecture diagram of Ansible [7]. The architecture consists of the following elements -

1. User - The users are system administrators, DevOps engineers, or IT professionals responsible for automating configuration and management tasks across servers or cloud resources. Users interact with Ansible by writing playbooks and maintaining inventories.
2. Control node – This is the node on which Ansible is installed. It initiates the machine setup on other machines. It contains Ansible's Automation Engine, playbooks, inventory, and configuration files. Multiple machines can be setup using Ansible at the same time.
3. Ansible Automation Engine - It is the software component installed on the control node to process and execute tasks. It is the core component which interprets playbooks and executes tasks mentioned in the playbook on managed nodes.
4. Managed nodes - The machines on which the setup is done by the control node are called managed or target nodes. Ansible uses an agentless model. It uses SSH (or WinRM for Windows) to communicate and execute tasks remotely.
5. Inventory - It is a file which contains list of IP addresses or hostnames of the managed nodes. The managed nodes may be organized into groups based on their roles or configurations. Inventory file can be in ini or yaml format.
6. Modules - Modules are small programs which are sent to managed nodes to execute tasks. There are 3 types of Ansible modules – Core modules, custom modules and third party modules.
7. Playbooks - These are the files written in yaml formal. It defines the tasks to run and the order in which the tasks should be run on the manages nodes. We can define the nodes on which the tasks should be run along with the associated variables.
8. Plugins – These are the modular components that extend and customize Ansible's core functionality. Plugins in Ansible are Python-based modules that hook into various parts of Ansible's execution engine.
9. Private/Public Cloud - Ansible can interact with both private and public cloud environments to automate the deployment and management of cloud-based infrastructure.

10. Configuration Management Database (CMDB) - It is a repository that stores information about IT assets, configurations and current and historic state.

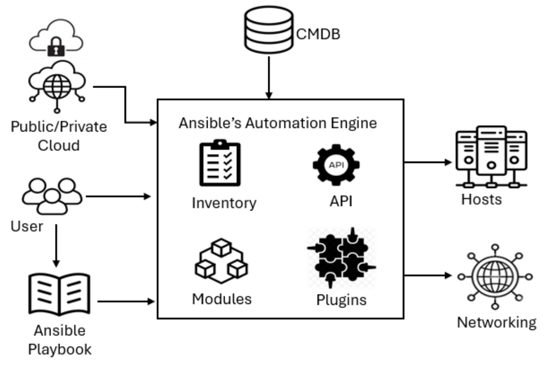

Fig. 1. Architecture diagram of Ansible.

B. *Various Scenarios and how Ansible handles them*

Changes in the build infrastructure or the machine setup are very frequent. Following are the various scenarios -

1. New machines are added in the pool of production machines - Addition of these machines is required in the inventory file. When the setup is triggered, the machine setup is done on all the managed nodes mentioned in the inventory file. Having all managed nodes in the inventory does not cause any issues due to the idempotency maintained by the Ansible backend. Idempotency means that running the same Ansible playbook or task multiple times will always produce the same result and would not cause unintended changes after the first successful run [8].

2. Version of a tool is changed - In this case, playbooks need to be updated for the required changes. When the setup is triggered, the desired version is installed on all the managed nodes.

3. New tool is required to be installed - Ansible backend needs to be updated if the support for new tool is not there. Once the backend is updated, the playbooks are then updated to get the new changes. When the setup is triggered, the new tool is installed on all the managed nodes present in the inventory file.

4. Different setup required on different set of machines – The managed nodes in the inventory file can be divided into groups. Each group should have nodes on which the same setup is required. These group names are mentioned in the host field of the playbooks. This helps in running the playbooks on the desired group of managed nodes and exclude from the groups on which the playbook does not need to be run.

C. *Resolution of manual setup issues using Ansible*

In table II we have shared the issues with the manual setup. Table III shows the ways how Ansible is able to resolve those manual issues.

TABLE III

SETUP ISSUES AND RESOLUTION USING ANSIBLE

Issue	How Ansible resolves it
Inconsistency Across Systems	Ansible employs idempotent playbooks to enforce consistent system state.
Lack of Documentation and Reproducibility	Ansible playbooks, maintained in version control systems like Git act as living documentation that can be executed at any time.
Scalability challenges	Ansible automates the installation and configuration processes across all hosts.
Configuration Drift	Regular execution of Ansible playbook ensures desired system state.
No Rollback or Version Control	Ansible Playbooks enable easy rollback of changes.
Security Risks	Ansible integrates secure repositories and validation steps.
Slower Recovery and Onboarding	Increased productivity as onboarding time for a new developer is very less.
Hard to Audit or Monitor	Logs and reports are generated from Ansible playbook runs.

III. EXPERIMENTS AND RESULTS

In this section, we describe the experimental setup for analyzing the machine setup time across varying cluster size using Ansible. The objective of the experiment is to evaluate the effectiveness and efficiency of automated deployment processes facilitated by Ansible in a controlled infrastructure. Our setup involved defining playbooks that automate the installation and configuration of software components across multiple nodes. We have used a Linux machine with configuration 4 vCPU, 16GB RAM and 100 GB Disk as the control node.

Ansible setup requires playbooks which define the desired state of the system. We maintained playbooks in a git repository for both Windows and Linux machines setup. Inventory file listing the nodes on which the setup needs to be done is maintained in a separate git repository. The Ansible control node executed these playbooks to manage the remote hosts without manual intervention using Ansible Web eXecutable (AWX) [9]. AWX is an open-source, web-based interface that includes a task execution engine, API, and web-based UI for executing, scheduling, and monitoring Ansible playbooks and jobs, thereby eliminating the need for manual intervention. It facilitates the management and automation of Ansible deployments. AWX supports the scaling of automation processes within complex environments by streamlining Ansible operations. It offers centralized control and enhanced visibility, delivers real-time updates on job statuses, detailed output logs, and notification features.

AWX provides Slice and Forks option. Slicing divides the inventory into the number mentioned by the users for parallel setup. For example, if there are 50 machines on which setup needs to be done and the slice value is set to 10 then 10 (number of slices) jobs are created, and each job takes care of the setup of 5 (total number of machines in inventory file/number of slices) machines. All these 10 jobs are run in parallel to complete the machine setup. The other option of fork handles the amount of parallelism in each job. If a job has an inventory of 10 machines and the fork value is set to 5 then the tasks mentioned in the playbooks are run in parallel on first

5 machines and then on the next set of 5 machines. Forks refers to the number of parallel processes Ansible uses to execute tasks on managed hosts during a job run. Increasing the fork count allows AWX to run tasks on multiple hosts simultaneously, speeding up deployments and reducing total runtime. The default fork value is typically 5, but it can be adjusted in AWX settings or per job template to optimize performance based on the environment and resources. For example, setting forks to 20 means AWX will run tasks on 20 hosts at the same time. Slice is a feature in AWX that allows you to split the execution of a job across multiple smaller batches (or slices) of hosts. Instead of running the job on all hosts at once, AWX will run it on a subset of hosts (a slice), wait for that batch to complete, then proceed to the next slice. This helps control the load on systems, reduce the risk of impacting all hosts simultaneously, and manage resources more effectively during large-scale deployments. For example, if you have 100 hosts and set the slice size to 20, AWX will run the job on 20 hosts at a time, completing each slice sequentially until all hosts are processed.

The experimental environment consists of clusters of Windows 10 enterprise and Ubuntu 24.04 version virtual machines with 4 vCPUs, 16 GB RAM and 100 GB storage. The playbook for Windows installed Visual Studio along with the corresponding .vsconfig file, as well as additional software packages including Notepad++, Google Chrome, 7-Zip, Windows SDK, CURL, PowerArchiver, Node.js, Java, Yarn, Grunt CLI, and Git. Furthermore, the playbook automated the update of registry entries and environment variables to ensure proper system configuration. The playbook for Ubuntu automated the installation of software packages such as Java, Yarn, Grunt CLI, Docker and Node.js.

Table IV shows setup time for clusters of varying sizes of Windows VMs and Table V shows setup time for cluster of varying sizes of Ubuntu VMs. We also compare the manual setup time with the setup time when Ansible is used, showing the gain achieved for Windows machines. Manual configuration involves individually executing commands, installing software, and configuring settings on each of the machines in the cluster. The setup time scales linearly with the number of machines, making it inefficient for large or complex environments.

To calculate the percentage gain on Windows machines, let N denote the total number of machines on which the setup was performed, P denote the number of machines on which the manual setup process can be executed concurrently, B denotes the number of setup batches and t_{manual} denotes the time taken for manual setup on 1 batch. P is much less than N as it is constrained by the availability of personnel and the overhead associated with coordinating parallel manual tasks.

TABLE IV

SETUP TIME ON WINDOWS VM

Number of Windows VMs	**Setup Time using Ansible**
1	45 min 18 sec
5	51 min 54 sec
10	46 min 18 sec
15	57 min 47 sec
20	52 min 45 sec
25	55 min 25 sec
30	1 hr 3 min
35	59 min 12 sec
40	1 hr 6 sec
45	59 min 42 sec
50	1hr 12 min

TABLE V

SETUP TIME ON UBUNTU VM

Number of Ubuntu VMs	**Time**
1	6 min 43 sec
5	6 min 48 sec
10	7 min 44 sec
15	7 min 58 sec
20	8 min 44 sec
25	8 min 43 sec
30	7 min 47 sec
35	9 min 16 sec
40	8 min 1 sec
45	9 min 54 sec
50	12 min 15 sec

Let T_{manual} and $T_{ansible}$ denote the total time taken for manual setup and setup using Ansible respectively on N machines. Let G denote the percentage of gain achieved when Ansible is used for the machine setup with forks and slice values set to 5 and 10 respectively.

$$N = 50 \tag{1}$$

$$P = 10 \tag{2}$$

$$B = N/P = 5 \tag{3}$$

$$t_{manual} = 1 \text{ hour} \tag{4}$$

$$\begin{aligned} T_{manual} &= B * t_{manual} \\ &= 5 * 1 \text{ hour} \\ &= 300 \text{ minutes} \end{aligned} \tag{5}$$

$$\begin{aligned} T_{ansible} &= 1 \text{ hour and } 12 \text{ minutes} \\ &= 72 \text{ minutes} \end{aligned} \tag{6}$$

$$Gain = T_{manual} - T_{ansible} \tag{7}$$

$= (300 - 72)$ *minutes*

$= 227$ *minutes*

$$G = (T_{manual} - T_{ansible}) / T_{manual} * 100 \quad (8)$$

$= (227/300) * 100$

$= \sim 75\%$

This is the gain we achieved in terms of time required for machine setup. It is the time duration after which the machines become usable. In actual sense, the gain is more than 75% because setup via Ansible is done with just few clicks which hardly takes seconds. While the setup is performed via Ansible, the user can focus on other tasks, whereas in the case of manual setup, the user is fully occupied and unable to engage in additional activities. The actual gain is likely greater, as Ansible setups can run anytime, while manual setups are limited to working hours, affecting engineer productivity.

IV. CONCLUSION

We evaluated the scalability of the deployment process using Ansible by measuring setup times, including environment configuration and the installation of multiple tools, across varying cluster sizes. The test environment consisted of homogeneous Windows and Linux virtual machines connected over a local network, thereby eliminating variability due to hardware differences or network latency. All machines were initialized simultaneously using Ansible's parallel execution settings, with the forks and slice parameters set to 5 and 10, respectively.

Despite a substantial increase in the number of deployment targets, the setup time exhibited only a marginal increase. This demonstrates Ansible's efficiency in parallel execution and highlights its suitability for managing large, distributed environments with minimal overhead. The near-linear scalability observed suggests that the Ansible framework maintains high efficiency even as system size increases. These results underscore its ability to deliver consistent performance at scale, which is critical in real-world scenarios where rapid and reliable infrastructure provisioning is essential. Ansible drastically reduces setup time as the environment grows in size or complexity. Automation through Ansible not only speeds up deployment but also improves reliability and maintainability.

REFERENCES

[1] S. Uzunbayir and K. Kurtel, "A Review of Source Code Management Tools for Continuous Software Development," 2018 3rd International Conference on Computer Science and Engineering (UBMK), Sarajevo, Bosnia and Herzegovina, 2018, pp. 414-419, doi: 10.1109/UBMK.2018.8566644.

[2] A. Tiwari, A. ., V. ., and A. Kumar, "Strategies for Pull Request Validation in CI/CD Practices," *Proceedings of the International Conference on Innovative Computing & Communication (ICICC 2024)*, Mar. 6, 2024. doi: 10.2139/ssrn.4749815.

[3] J. Alam, R. Haque, T. Akter, and F. Nishi, "Multi-OS Configuration Drift Detection Using Ansible," *ResearchGate*, 2022.

[4] M. Gupta, M. N. Chowdary, S. Bussa and C. K. Chowdary, "Deploying Hadoop Architecture Using Ansible and Terraform," 2021 5th International Conference on Information Systems and Computer Networks (ISCON), Mathura, India, 2021, pp. 1-6, doi: 10.1109/ISCON52037.2021.9702299.

[5] E. Ozdogan, O. Ceran, and M. T. Ustundag, "Systematic Analysis of Infrastructure as Code Technologies," *Gazi University Journal of Science Part A: Engineering and Innovation*, vol. 10, no. 4, pp. 452–471, 2023, doi: 10.54287/gujsa.1373305.

[6] A. B. B. Ojel and J. I. Teleron, "Configuration Management and Automation Tools: A Comparative Analysis and Overview," International Journal of Advanced Research in Arts, Science, Engineering & Management (IJARASEM), vol. 12, no. 1, pp. 1–10, Jan.–Feb. 2025.

[7] A. Cepuc, R. Botez, O. Craciun, I. -A. Ivanciu and V. Dobrota, "Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes," *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, Bucharest, Romania, 2020, pp. 1-6, doi: 10.1109/RoEduNet51892.2020.9324857.

[8] Ansible Documentation, "Desired State and Idempotency," *Ansible*, [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html#desired-state-and-idempotency.

[9] Ansible AWX Documentation, "*AWX Project Documentation*," [Online]. Available: https://ansible.readthedocs.io/projects/awx/en/latest/.