

Week -9

Chef

What is Chef?

Chef is a:

Configuration Management Tool
Infrastructure Automation Tool
Infrastructure as Code (IaC) tool

Simple Definition

Chef is used to **automate the installation and configuration of software on servers using code.**

Instead of manually installing software on each server, we write code (recipes), and Chef installs automatically.

Why Do We Need Chef?

Imagine you have 10 AWS EC2 servers.

Without chef You do manually:

- Install Java
- Install Nginx
- Install Tomcat
- Configure services
- Repeat on all servers

This is:

- Time consuming
- Error-prone
- Not scalable

With Chef

You write one recipe:

package 'nginx'

service 'nginx'

Chef installs automatically on all servers.

What Problem Does Chef Solve?

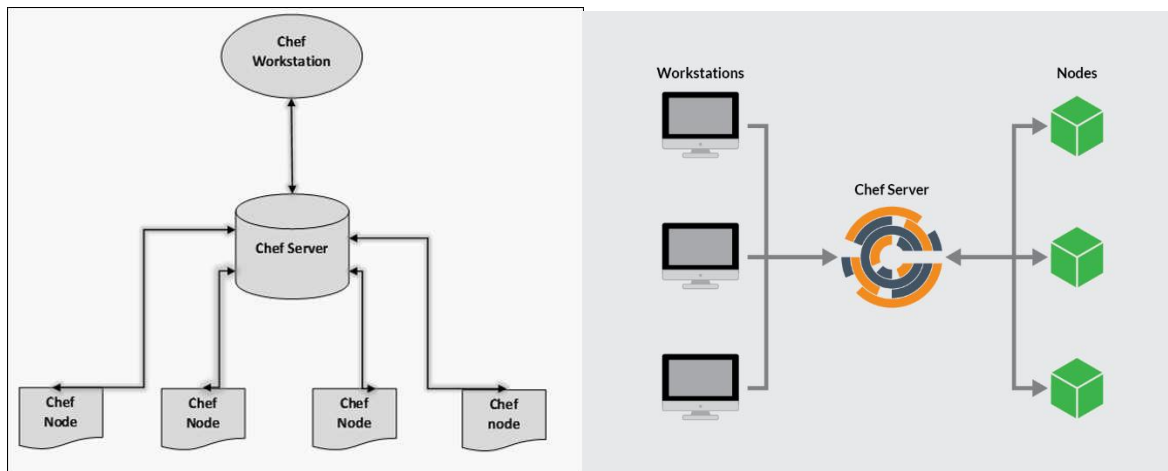
Chef solves:

Manual configuration problem
Environment inconsistency
Scaling problems
Human errors
Repetitive tasks

How Chef Works (Architecture):

There are 2 main models:

1. **Chef Server Model (Enterprise)**



Components:

Component	Role
Chef Server	Stores configuration
Chef Workstation	Where cookbooks are written
Chef Client	Runs on nodes
Node	Managed server (EC2)

Flow:

Workstation → Upload cookbook → Chef Server

Node → Pull configuration → Apply

This is called:

Pull Model

2.Chef Solo (Local Mode)

Simpler version:

- No Chef Server
- Recipes stored locally
- Run using:

`chef-client --local-mode`

Best for:

- Learning
- Small projects
- 2–5 servers

Important Chef Terminology

Cookbook

Collection of recipes.

Example:

- `nginx_cookbook`
- `java_cookbook`

Recipe

Instructions written in Ruby.

Example:

```
package 'nginx' do
  action : install
end
```

Resource

Basic unit of configuration.

Examples:

- package
- service
- file
- user

Run List

Defines which recipes run on a node.

Example:

```
recipe[java_cookbook]
recipe[nginx_cookbook]
```

Example Scenario

Suppose:

- EC2-1 → Install Java
- EC2-2 → Install Nginx

Chef process:

1. Create Java cookbook
2. Create Nginx cookbook
3. Assign run list
4. Run chef-client

Automation completed.

What Happens Internally?

When you run:

```
sudo chef-client
```

Chef:

```
Contacts Chef Server (if using server mode)
Downloads cookbook
Checks system state
Installs missing packages
Starts services
Reports status
```

Chef ensures:

Idempotency

Means: Running multiple times does not brake system.

Advantages of Chef

- Automation
- Consistency
- Scalable
- Infrastructure as Code
- Reusable cookbooks
- Enterprise-ready

Disadvantages:

- Complex setup
- Ruby knowledge helpful
- Chef Server heavy for small projects

Chef vs Manual Configuration

Manual	Chef
Login each server	Automated
Install manually	Code-based
Risk of mistake	Consistent
Hard to scale	Easy to scale

Configuration Management Using Chef Solo

1. Aim:

To configure multiple AWS EC2 instances using **Chef Solo (Chef Infra Client in local mode)** by installing:

- Nginx on one server
- Java on another server

2.Objective:

To automate software installation and configuration using Chef without setting up a Chef Server.

Tools & Technologies Used

- Amazon Web Services
- Ubuntu 22.04 EC2 instances
- Chef
- Nginx
- OpenJDK 17

3. Architecture Flow (Conceptual):

EC2 Instance	Role
EC2-1	Chef Workstation
EC2-2	Nginx Node
EC2-3	Java Node

Flow:

```
graph TD
    A[Chef Workstation] -- "(SCP)" --> B[Nodes]
    B --> C[Run chef-client --local-mode]
```

Pre-Requisites

- AWS account
- 3 EC2 Ubuntu instances
- Same key.pem file

Ports opened:

Server	Ports
Workstation	22
Nginx Node	22, 80
Java Node	22

4. Procedure:

Step 1: Install Chef Workstation

Login to workstation:

```
ssh -i ubuntu@<workstation-ip>
```

Install Chef:

```
curl -LO https://omnitruck.chef.io/install.sh
sudo bash install.sh -P chef-workstation
```

Verify:

```
chef --version
```

(Note: Chef Workstation is the **control machine** where:

- You write cookbooks
- Create recipes
- Test configurations

It contains tools like:

- chef
- knife
- cook style
- Workstation creates cookbooks.
- Nodes execute them using `chef-client --local-mode`.)

Step 2: Install Chef Client on Nodes

(Note:

We install Chef Client only (not workstation tools) on:

- Nginx Node
- Java Node

Chef Client

This is the software that executes cookbooks)

On nginx-node:

```
curl -LO https://omnitruck.chef.io/install.sh
sudo bash install.sh -P chef
```

On java-node:

```
curl -LO https://omnitruck.chef.io/install.sh
sudo bash install.sh -P chef
```

Step 3: Create Cookbooks on Workstation

On workstation:

```
chef generate cookbook nginx_cookbook
chef generate cookbook java_cookbook
```

(Note:

A cookbook is a folder containing,

- Recipes
- Attributes
- Templates
- Files

Think of cookbook like:

A project folder for automating one software.

Cookbook Structure:

```
nginx_cookbook/
├── recipes/
│   └── default.rb
├── metadata.rb
├── attributes/
└── templates/
```

)

Step 4: Configure Nginx Recipe

Edit:

```
nano nginx_cookbook/recipes/default.rb
```

Add:

```
package 'nginx' do
  action :install
end
```

```
service 'nginx' do
```

```

        action [:enable, :start]
      end
    end
  end
end

```

Save file.

(Note:

- package → Chef resource
- 'nginx' → package name
- :install → tells Chef to install
- :enable → starts on boot
- :start → starts immediately)

Step 5: Configure Java Recipe

Edit:

```
nano java_cookbook/recipes/default.rb
```

Add:

```

package 'openjdk-17-jdk' do
  action :install
end

```

Save file.

Step 6: Copy Cookbooks to Nodes:

You are **not using a .pem file**, so the only way for two EC2 instances to connect is **SSH key sharing between instances**. Right now the **workstation** is trying to connect to **Java node** but the Java node **does not trust the workstation's SSH key**.

So, you must **copy the workstation public key to the Java node**.

When you use **EC2 Instance Connect**, AWS temporarily adds a key only for **browser login**. But **instance-to-instance communication needs permanent SSH keys**, so you must manually add the key to `authorized_keys`.

After doing the below process once, **scp / ssh between instances will always work without a .pem file**.

1. Generate SSH key on Workstation

On the **workstation (172.31.71.126)** run:

```
ssh-keygen
```

Press **Enter 3 times**.

This creates:

```
~/.ssh/id_ed25519
```

```
~/.ssh/id_ed25519.pub
```

2. Display the public key

Run:

```
cat ~/.ssh/id_ed25519.pub
```

Example output:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI... ubuntu@ip-172-31-71-126
```

Copy the **entire line**.

3. Login to Java Node

Open **Java node terminal using EC2 Instance Connect**.

You will see something like:
ubuntu@ip-172-31-73-72

4. Add key to Java Node

Run:

```
mkdir -p ~/.ssh  
nano ~/.ssh/authorized_keys
```

Paste the **copied key**.

Example:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI... ubuntu@ip-172-31-71-126
```

Save:

```
CTRL + X  
Y  
Enter
```

5. Fix permissions on java node

Run:

```
chmod 700 ~/.ssh  
chmod 600 ~/.ssh/authorized_keys
```

6. Test SSH connection

Go back to **workstation** and run:

```
ssh ubuntu@172.31.73.72
```

If it logs in without error, SSH is fixed.

7. Now copy the cookbook

Run from **workstation**:

```
scp -r java_cookbook ubuntu@172.31.73.72:/home/ubuntu/
```

From workstation:

```
scp -r nginx_cookbook ubuntu@<nginx-ip>:/home/ubuntu/  
scp -r java_cookbook ubuntu@<java-ip>:/home/ubuntu/
```

(Note:

scp → secure copy

-r → recursive (copy entire folder)

You are manually transferring cookbook because:

We are NOT using Chef Server

This is Local Mode (like Chef Solo))

Step 7: Run Chef in Local Mode

(Note: Even though you are inside the folder, Chef cannot find the cookbook directory structure because local-mode expects a cookbooks directory.

Right now, your structure on the node is probably like this:

```
/home/ubuntu/  
  java_cookbook/  
    recipes/  
      metadata.rb
```

But Chef local-mode expects this structure:

```
/home/ubuntu/  
  cookbooks/  
    java_cookbook/  
      recipes/  
        metadata.rb
```

So Chef is not locating the cookbook.

So, now we have to fix above error at java node

Step 1 — Go back to home

```
cd ~
```

Step 2 — Create cookbooks folder

```
mkdir cookbooks
```

Step 3 — Move your cookbook

```
mv java_cookbook cookbooks/
```

Now structure becomes:

```
/home/ubuntu/cookbooks/java_cookbook
```

Step 4 — Run Chef again

```
sudo chef-client --local-mode -o recipe[java_cookbook]
```

If u get error, Run APT update first on the Java node.

```
sudo apt update
```

On Nginx Node:

```
sudo chef-client --local-mode --runlist 'recipe[nginx_cookbook]'
```

On Java Node:

```
sudo chef-client --local-mode -o recipe[java_cookbook]
```

(Note1: chef-client → runs chef

--local-mode → no chef server

--runlist → which recipe to run

recipe[nginx_cookbook] means:

Execute default.rb inside nginx_cookbook)

Note2:

(First You Copied Cookbook:

```
scp -r nginx_cookbook ubuntu@<nginx-ip>:/home/ubuntu/
```

At this stage:

- Nothing is installed
- Node has only files

No configuration applied yet.

Next Node Runs chef-client:

On node:

```
sudo chef-client --local-mode --runlist 'recipe[nginx_cookbook]'
```

Now observe carefully

Node itself started Chef

Node reads cookbook

Node checks system state

Node installs nginx

Workstation did NOT execute installation.

Node executed it.

That is Pull behaviour.)

(**Note3:** Why it is called pull?

Because:

Node pulls configuration instructions and applies them.

Even in Local Mode:

- Node pulls recipe from local directory
- In Chef Server model, node pulls from Chef Server

In this setup, Chef worked in Pull Model because:

1. Chef Client runs on target node.
2. Node initiates configuration process.
3. Node reads and applies cookbook.
4. Workstation does not directly install software.
5. Configuration execution happens from node side.

)

Step 7: Verification

Verify Nginx

Open browser:

http://<nginx-ip>

Nginx default page should appear.

Verify Java

On java-node:

```
java -version
```

Java version displayed successfully.

5. Result:

Successfully configured:

- Nginx on EC2-2
- Java on EC2-3

Using Chef Solo without Chef Server.

Automation completed successfully.

6.Conclusion:

Chef Solo simplifies configuration management for small-scale environments by allowing local execution of recipes without requiring centralized infrastructure.

Week 10

Puppet

What is Puppet?

Puppet develops **Puppet**.

Puppet is a:

- Configuration Management Tool
- Infrastructure Automation Tool
- Infrastructure as Code (IaC) tool

It is used to **automatically install and configure software on multiple servers**.

Simple Definition

Puppet is a tool that manages servers automatically using code instead of manual configuration.

Why Do We Need Puppet?

Suppose you are using:

Amazon Web Services

You have 10 EC2 servers.

You want:

- All servers → Java installed
- One server → Nginx
- One server → Tomcat

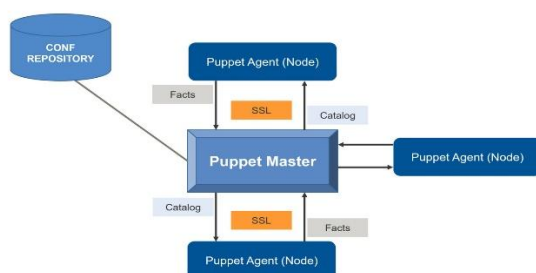
Without Puppet

You log in manually to each server and configure.

With Puppet

You write configuration code once, and Puppet automatically applies it to all servers.

How Puppet Works (Architecture)



Puppet follows a **Pull Model**.

Components:

Component	Role
Puppet Master	Central configuration server
Puppet Agent	Installed on nodes
Node	Managed server
Manifest	Configuration file (.pp file)

Puppet Workflow:

Puppet Agent runs on node
Agent contacts Puppet Master
Master sends configuration (catalog)
Agent applies configuration
Agent reports back

Default interval:

Every 30 minutes (configurable).

Important Puppet Terminology:

1. Manifest

Configuration file written in Puppet DSL (.pp file)

Example:

```
package { 'nginx':  
  ensure => installed,  
}
```

```
service { 'nginx':  
  ensure => running,  
  enable => true,  
}
```

2. Resource

Basic building block.

Examples:

- package
- service
- file
- user

3. Node

Any managed server (EC2 instance).

4. Catalog

Compiled configuration sent from Master to Agent.

Example (Your Scenario):

You have 3 EC2 instances:

Server	Requirement
EC2-1	Java
EC2-2	Java + Nginx
EC2-3	Java

Step 1: Install Puppet Master on EC2-1

Step 2: Install Puppet Agent on EC2-2 & EC2-3

Step 3: Create manifest file:

```
node 'ec2-2' {  
  package { 'nginx':  
    ensure => installed,  
  }  
}  
  
node default {  
  package { 'openjdk-17-jdk':  
    ensure => installed,  
  }  
}
```

Now agents pull configuration automatically.

What Happens Internally?

When Puppet Agent runs:

```
sudo puppet agent -t
```

Internally:

- Agent sends system facts (OS, IP, memory)
- Master compiles catalog
- Agent compares current state
- Installs missing software
- Starts services
- Sends report

Puppet ensures:

Idempotency

It will not reinstall if already installed.

Advantages of Puppet

- Centralized control
- Enterprise-ready
- Secure SSL communication
- Scalable to thousands of servers
- Strong compliance management

Disadvantages

- Complex setup
- Requires Puppet Master
- Learning curve

Puppet vs Chef (Quick View)

Feature	Puppet	Chef
Language	Puppet DSL	Ruby

Feature	Puppet	Chef
Model	Pull	Pull
Complexity	Medium	Medium/High
Enterprise	Yes	Yes

Push vs Pull (Important)

Tool	Model
Ansible	Push
Chef	Pull
Puppet	Pull

Puppet = Node pulls configuration from master.

Push vs Pull

Push Model

Control server **pushes configuration** to nodes.

Meaning:

“I am the main server. I will send instructions to all servers.”

Pull Model

Nodes **pull configuration** from central server.

Meaning:

“I am a server. I will go and ask the master for configuration.”

Push Model (Example: Ansible)

Tool example:

Red Hat (Ansible)

How It Works:

Control Node

↓ (SSH push)

Server1

Server2

Server3

Admin runs one command:

`ansible-playbook install_java.yml`

Control node immediately sends instructions.

No agent required.

Pull Model (Example: Chef & Puppet)

Tool examples:

Chef

Puppet

How It Works:

Master Server
↓
Nodes (with agent installed)
Each node runs:
 chef-client
Node contacts master and pulls configuration.

Visual Comparison

Push Model

Control Node
↓↓↓
Server1
Server2
Server3
Admin controls everything.

Pull Model

Server1 →
Server2 → Master
Server3 →
Servers request configuration.

Real Example (Your AWS Case)

Using:
Amazon Web Services
You have 10 EC2 instances.

If Using Push Model (Ansible)

From one Jenkins server:
 ansible-playbook deploy.yml
Immediately configures all servers.
Good for:

- Small setup
- Fast deployment

If Using Pull Model (Chef)

Each EC2 runs:
 chef-client
Server checks:
 Is Java installed?
 Is Tomcat running?

Good for:

- Large infrastructure
- Continuous configuration enforcement

key Differences Table

Feature	Push Model	Pull Model
Who starts action?	Control Server	Node
Agent required?	No (usually)	Yes
Example	Ansible	Chef, Puppet
Real-time control	Immediate	Based on schedule
Best for	Small/Medium setup	Large enterprise

Important Interview Questions

Q: Which model is more scalable?

Answer:

Pull model is more scalable because each node independently pulls configuration from master server.

Q: Why is pull model secure?

Answer:

Because nodes initiate connection to master, reducing direct external access.

Simple Understanding

Push = Teacher gives homework

Pull = Student asks teacher for homework

Configuration Management Using Puppet

1. Aim:

To configure multiple AWS EC2 instances using Puppet by installing:

- Java on one server
- Nginx on another server

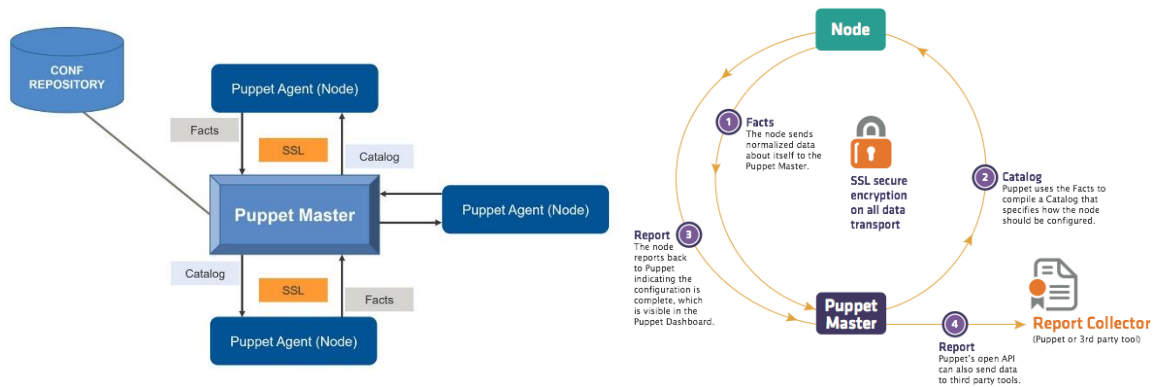
2. Objective:

To understand Puppet architecture and implement configuration management using the Master-Agent model.

3. Tools & Technologies Used:

- Amazon Web Services
- Ubuntu 22.04 EC2 Instances
- Puppet
- Nginx
- OpenJDK 17

4. Architecture Diagram



Components:

Component	Description
Puppet Master	Central configuration server
Puppet Agent	Installed on nodes
Node	Managed EC2 instance
Manifest	Configuration file (.pp)

5. Pre-Requisites:

- AWS Account
- 3 Ubuntu EC2 Instances
- Same Key Pair
- Ports Opened:

Server	Ports
Puppet Master	22, 8140
Java Node	22
Nginx Node	22, 80

6.Procedure:

Step 1: Install Puppet Server (Master Node)

Login to Master:

```
ssh -i key.pem ubuntu@<master-ip>
```

Install Puppet Server:

```
sudo apt update
```

```
sudo apt install puppetserver -y
```

Start service:

```
sudo systemctl start puppetserver
```

```
sudo systemctl enable puppetserver
```

Fix Hostname Resolution (Important)

Edit hosts file on **Master**:

```
sudo nano /etc/hosts
Add:
    127.0.0.1 localhost
    172.31.76.162 puppet ip-172.31.76.162
Test:
    ping puppet
```

Step 2: Install Puppet Agent (On Java & Nginx Nodes)

(Note: In Puppet's Pull Model, nodes must have an agent installed.
The agent contacts the Puppet Master and pulls configuration)

```
sudo apt update
sudo apt install puppet-agent -y
Edit configuration on node:
sudo nano /etc/puppet/puppet.conf
Add:
    [main]
    server = puppet
```

(Note: We must tell the agent, Which Puppet Master to connect to. Agent will contact Puppet Master at that IP.

Used IP instead of hostname Why?

- Puppet SSL certificates use **hostname**
- Using IP causes **certificate mismatch errors**)

Add Master Hostname Mapping

On Java and Nginx nodes:

```
sudo nano /etc/hosts
Add:
    172.31.78.6 puppet
Start agent:
sudo systemctl start puppet
sudo systemctl enable puppet
```

Step 3: Approve Node Certificates on Master

(Note: Why Certificate?

Puppet uses SSL authentication for secure communication.

When agent first connects:

It sends a certificate request.

Master must approve it)

On Master:

```
sudo puppetserver ca list
```

(Note:

Output:

"java-node" (SHA256) ...

"nginx-node" (SHA256) ...

This means:
Both nodes are requesting trust.)

Approve:
sudo puppetserver ca sign --all

(Note: **What it does:**
Approves all pending nodes.
Now nodes are trusted.
After approval:
Secure communication established
Agent can pull configurations)

Step 4: Create Manifest File on Master

Create Puppet Environment

Create directory:

```
sudo mkdir -p /etc/puppet/code/environments/production/manifests
```

Create Manifest File

Open:

```
sudo nano /etc/puppet/code/environments/production/manifests/site.pp
```

(Note: **What is site.pp?**
It is Puppet's **main manifest file**.
It defines configurations for all nodes.
Think of it like:
Main configuration script.)

Add:

```
# Java Server Configuration
node default {
  package { 'openjdk-17-jdk':
    ensure => installed,
  }
}

# Nginx Server Configuration
node 'nginx-node' {
  package { 'nginx':
    ensure => installed,
  }

  service { 'nginx':
    ensure => running,
    enable => true,
  }
}
```

(Note: If node name = java-node

→ Install OpenJDK 17

If not installed → Install

If installed → Do nothing

Idempotent behavior (Very Important in Puppet)

Idempotency:

If you run Puppet 100 times:

- It will not reinstall nginx again.
- It maintains desired state.

That is the power of Puppet.)

After this, **Restart Puppet Server**

On **master**:

```
sudo systemctl restart puppetserver
```

Step 5: Run Puppet Agent(on Nodes)

On Java Node:

```
sudo puppet agent -t
```

On Nginx Node:

```
sudo puppet agent -t
```

(Note: **What happens internally?**

1. Agent contacts Master
2. Downloads catalogue (configuration instructions)
3. Applies changes
4. Sends report back)

7. Verification:

Verify Java Installation

```
java -version
```

Java installed successfully.

Verify Nginx

Open browser:

```
http://<nginx-ip>
```

Default Nginx page appears.

8.Output:

Puppet Agent output shows:

Notice: Applied catalogue in X seconds

(Note: **Complete Flow Summary:**

Agent Installed

Agent Configured to point to Master

Certificate request sent

Master approves certificate

Master defines desired configuration

Agent pulls configuration
Software installed & services started)

9. Result:

Successfully configured:

- Java on one EC2 instance
- Nginx on another EC2 instance

Using Puppet Master–Agent architecture.

10. Conclusion:

Puppet simplifies infrastructure management using centralized configuration. It follows a pull model where agents retrieve configuration from the master and apply the desired state automatically.

Week 11

Ansible

What is Ansible?

Red Hat owns and maintains Ansible.

Ansible is an automation and configuration management tool.

It is used to:

- Install software
- Configure servers
- Deploy applications
- Automate tasks
- Manage cloud infrastructure

Simple Definition

Ansible is a tool that automates server configuration using SSH without installing agents.

Why Do We Need Ansible?

Suppose you are using:

Amazon Web Services

You have 20 EC2 servers.

Without Ansible

- Login into each server
- Install Java manually
- Install Nginx manually
- Configure settings
- Repeat 20 times

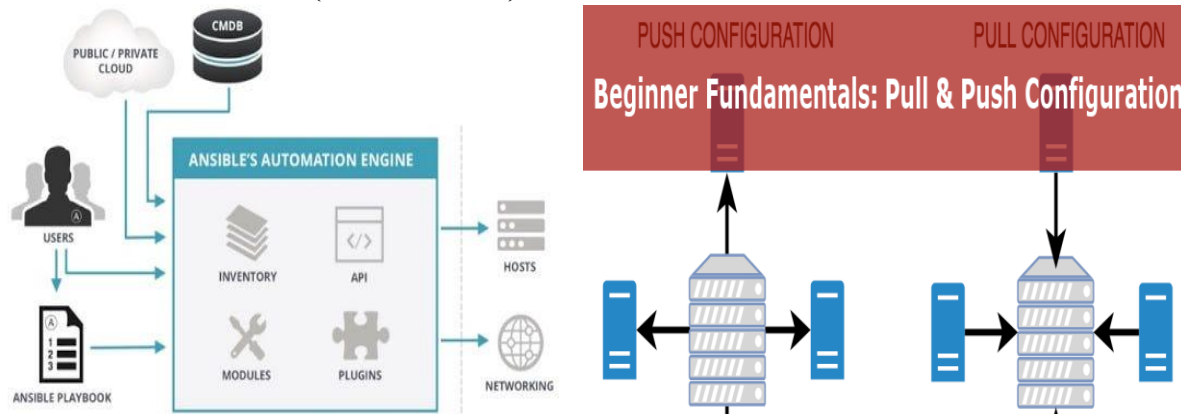
With Ansible

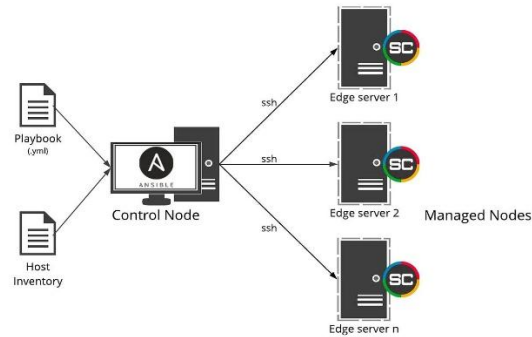
You write one playbook and run:

`ansible-playbook setup.yml`

All servers are configured automatically.

How Ansible Works (Architecture)





Ansible follows **Push Model**.

Components

Component	Role
Control Node	Where Ansible is installed
Managed Nodes	Target servers (EC2)
Inventory	List of servers
Playbook	YAML configuration file
Module	Small task unit (apt, service, copy, etc.)

Important Features

Agentless (no installation on nodes)

Uses SSH

Simple YAML syntax

Idempotent

Fast setup

Important Ansible Concepts

Inventory File

Defines server list.

Example:

```
[java_servers]
```

```
192.168.1.10
```

```
[nginx_servers]
```

```
192.168.1.20
```

Playbook

YAML file containing instructions.

Example:

```
- name: Install Java
```

```
  hosts: java_servers
```

```
  become: yes
```

```
  tasks:
```

- name: Install OpenJDK
apt:
 name: openjdk-17-jdk
 state: present

Task

Single action.

Example:

- Install package
- Start service
- Copy file

Module

Built-in function.

Examples:

- apt
- yum
- service
- copy
- file

become: yes

Runs tasks as root (sudo).

How Ansible Executes Internally

When you run:

ansible-playbook setup.yml

Internally:

Reads inventory

SSH connects to servers

Transfers small Python module

Executes task

Checks state

Returns result

It does NOT install any agent.

Example (Your Scenario)

You have 3 EC2:

Server	Requirement
EC2-1	Java
EC2-2	Nginx
EC2-3	Java

Inventory:

```
[java_servers]
ec2-1
ec2-3
```

```
[nginx_server]
ec2-2
```

Playbook:

```
- name: Install Java
  hosts: java_servers
  become: yes
  tasks:
    - apt:
      name: openjdk-17-jdk
      state: present

- name: Install Nginx
  hosts: nginx_server
  become: yes
  tasks:
    - apt:
      name: nginx
      state: present
```

Run:

```
ansible-playbook -i inventory.ini setup.yml
```

Automation completed

Idempotency (Very Important)

If you run playbook again:

It will NOT reinstall.

It checks:

Is Java already installed?

If yes → do nothing.

This is called **Desired State Configuration**.

Advantages of Ansible

Very easy to learn

No agent required

Minimal setup

Good for cloud environments

Widely used in DevOps

Disadvantages

Not ideal for extremely large infra compared to Puppet/Chef
Requires SSH access
Performance depends on network

Ansible vs Others

Tool	Model	Agent
Ansible	Push	No
Chef	Pull	Yes
Puppet	Pull	Yes

Where Ansible Is Used?

- CI/CD pipelines
- AWS automation
- Docker/Kubernetes setup
- Infrastructure provisioning
- Application deployment

Configuration Management Using Ansible

1.Aim:

To configure multiple AWS EC2 instances using Ansible by installing:

- Java on two servers
- Nginx on one server

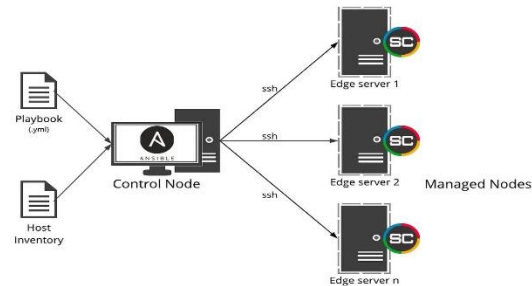
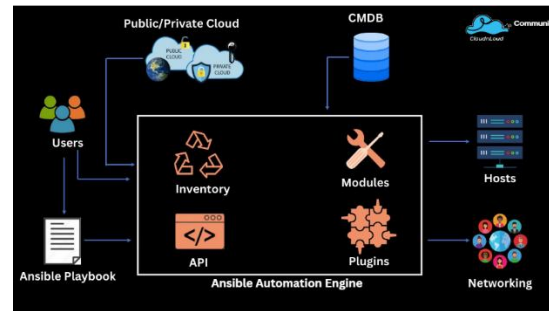
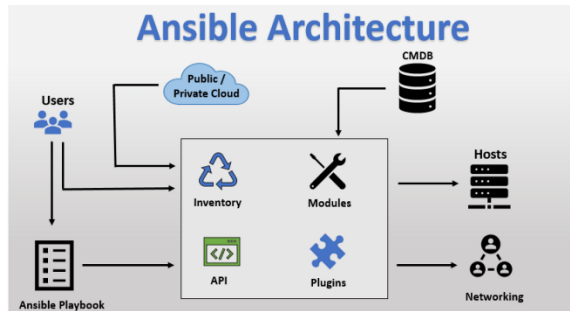
2. Objective:

To automate software installation and configuration using Ansible's push-based architecture.

3.Tools & Technologies Used:

- Amazon Web Services
- Ubuntu 22.04 EC2 Instances
- Red Hat (Ansible)
- Nginx
- OpenJDK 17

4.Architecture Diagram:



Components:

Component	Description
Ansible Master	Control node where Ansible is installed
Managed Nodes	Target EC2 instances
Inventory	List of servers grouped logically
Playbook	YAML configuration file

5.Pre-Requisites:

- AWS Account
- 4 Ubuntu EC2 Instances
- Same Key Pair
- Ports Opened:

Server	Ports
Master	22
Java Nodes	22
Nginx Node	22, 80

6.Procedure:

Step 1: Install Ansible on Master Node

Login to Master:

```
ssh -i key.pem ubuntu@<master-ip>
```

Install Ansible:

```
sudo apt update
```

```
sudo apt install ansible -y
```

Verify installation:

```
ansible --version
```

Step 2: Test SSH Connectivity

From Master:

```
ssh ubuntu@<java1-ip>
```

```
ssh ubuntu@<java2-ip>
```

```
ssh ubuntu@<nginx-ip>
```

Ensure SSH access works without password issues.

(The error:

ubuntu@172.31.69.34: Permission denied (publickey)

means **Instance 1 cannot authenticate to Instance 2 using SSH key**. Since you said you are using the **same .pem file on your computer**, the problem is that the **EC2 instance itself does not have the key**.

In AWS, **instances do NOT automatically share the pem file**, so you must copy it to the master instance.

Step-by-Step Fix

1. Copy PEM file from your computer to the first instance

Run this **from your local computer terminal** (not inside EC2):

```
scp -i key.pem key.pem ubuntu@<master-public-ip>:/home/ubuntu/
```

Example:

```
scp -i mykey.pem mykey.pem ubuntu@54.210.xx.xx:/home/ubuntu/
```

2. Login to the first instance

```
ssh -i key.pem ubuntu@<master-public-ip>
```

Check if pem exists:

```
ls
```

You should see:

```
key.pem
```

3. Change permission of pem file

Very important:

```
chmod 400 key.pem
```

4. SSH to second instance using the pem file

Now run:

```
ssh -i key.pem ubuntu@172.31.69.34
```

If private networking is enabled, it should work.)

Step 3: Create Inventory File on master

Create inventory file:

```
nano inventory.ini
```

Add:

```
[java_servers]
<java1-ip>
<java2-ip>
```

```
[nginx_server]
<nginx-ip>
```

(Note: Inventory file defines:
Target machines
Grouping of servers)

Step 4: Create Playbook File on master

nano setup.yml

Add:

```
- name: Install Java on Java servers
  hosts: java_servers
  become: yes
  tasks:
    - name: Install OpenJDK 17
      apt:
        name: openjdk-17-jdk
        state: present

- name: Install Nginx on Nginx server
  hosts: nginx_server
  become: yes
  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present

- name: Start nginx
  service:
    name: nginx
    state: started
    enabled: yes
```

(Note: Playbook is:
A YAML file
Contains automation steps
Defines what to install and where)

name → Description
hosts → Target group from inventory
become: yes → Use sudo privilege
state: started → Start service now
enabled: yes → Start on boot)

Step 5: Execute Playbook

Run:

```
ansible-playbook -i inventory.ini setup.yml --private-key sai.pem
```

(Note:

ansible-playbook → Runs playbook
-i inventory.ini → Specifies inventory file
setup.yml → Playbook file)

7.Verification:

Verify Java Installation

On both Java servers:

```
java -version
```

Java should be installed successfully.

Verify Nginx Installation

Open browser:

```
http://<nginx-ip>
```

Default Nginx page should appear.

8. Output:

At the end of execution, Ansible displays:

PLAY RECAP

Showing:

- ok = tasks already correct
- changed = modifications applied
- failed = errors (if any)

(Note: **What Happens Internally?**

1. Master reads inventory
2. Master connects to Java1 via SSH
3. Master installs Java
4. Master connects to Java2
5. Installs Java
6. Connects to Nginx server
7. Installs and starts nginx

All execution commands are sent from Master.

That is why Ansible is:

Push model)

9.Result:

Successfully installed:

- Java on two EC2 instances
- Nginx on one EC2 instance

Using Ansible push-based configuration management.

10.Conclusion:

Ansible simplifies infrastructure management by pushing configuration from a central control node to managed nodes using SSH and YAML playbooks.

Week 12

Salt Stack

What is SaltStack?

SaltStack developed **SaltStack**.

SaltStack is a:

- ◆ Configuration Management Tool
- ◆ Remote Execution Tool
- ◆ Infrastructure Automation Tool

It is used to:

- Install software
- Configure servers
- Manage large-scale infrastructure
- Execute commands on multiple systems

Simple Definition

SaltStack is a fast automation tool that manages servers using a master-minion architecture and supports both push and pull models.

Why SaltStack?

Suppose you are using:

Amazon Web Services

You have 100 EC2 instances.

You want:

- All servers → Java installed
- One server → Nginx
- One server → Tomcat

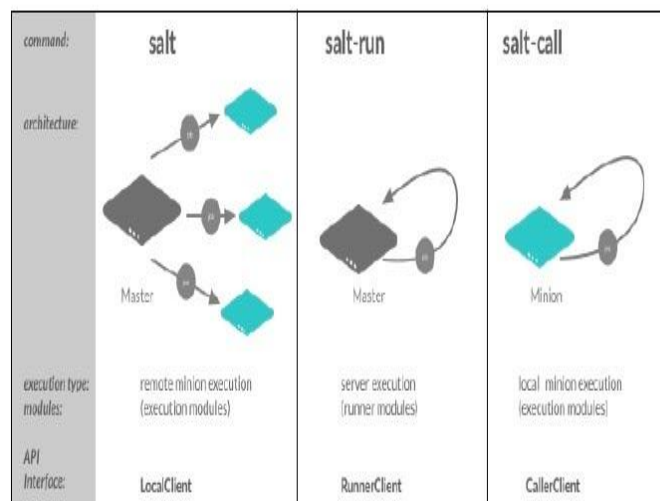
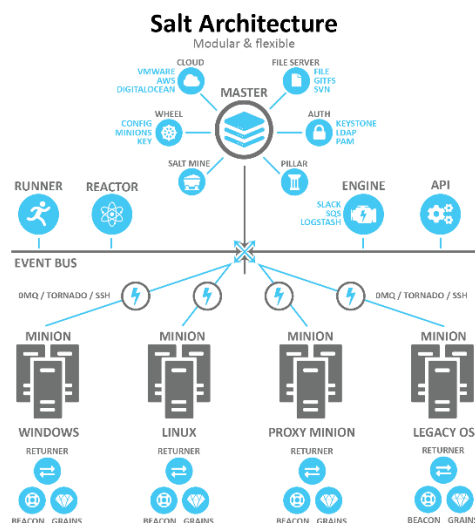
Salt can configure all of them **very fast**.

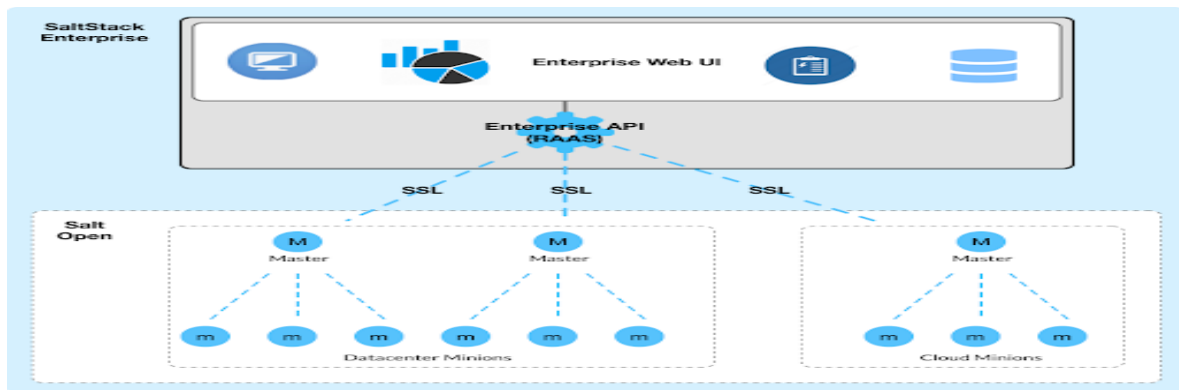
Salt is known for:

High speed

Real-time execution

SaltStack Architecture:





Components:

Component	Role
Salt Master	Central control server
Salt Minion	Installed on managed nodes
State File	Configuration file (.sls)
Pillar	Secure variable data

How SaltStack Works:

Salt uses:

Master–Minion Model

Master sends command
Minions execute
Results sent back

Salt uses **ZeroMQ messaging system** → Very fast communication.

SaltStack Models:

Push Model

Master pushes commands instantly.
Example:
`salt '*' test.ping`
Master sends command to all minions.

Pull Model

Minions request configuration from master.
Example:
`salt '*' state.apply`

Important Salt Terminology:

Minion

Managed server (like EC2 instance).

State File (.sls)

Configuration file written in YAML.

Example:
`install_java:`
`pkg.installed:`
`- name: openjdk-17-jdk`

Pillar

Secure configuration data (like passwords).

Grains

System information (OS, IP, memory).

Example (Your Scenario)

Suppose:

Server	Requirement
EC2-1	Java
EC2-2	Nginx
EC2-3	Java

Step 1: Install Salt Master on EC2-1

Step 2: Install Salt Minion on EC2-2 & EC2-3

Example State File

For Java:

```
install_java:
  pkg.installed:
    - name: openjdk-17-jdk
```

For Nginx:

```
install_nginx:
  pkg.installed:
    - name: nginx
```

```
start_nginx:
  service.running:
    - name: nginx
    - enable: True
```

Apply:

```
salt 'ec2-2' state.apply
```

What Makes Salt Different?

Extremely fast (parallel execution)

Real-time remote command execution

Supports Push & Pull

Scales to thousands of nodes

Salt vs Others

Tool	Model	Speed	Agent
Ansible	Push	Medium	No
Chef	Pull	Medium	Yes
Puppet	Pull	Medium	Yes
SaltStack	Push + Pull	Fast	Yes

Advantages of SaltStack

High performance

Real-time communication

Scalable
Good for large infrastructure

Disadvantages

Slightly complex
Requires master-minion setup
Less popular than Ansible in small setups

Configuration Management Using SaltStack

1.Aim:

To configure multiple AWS EC2 instances using SaltStack by installing:

- Java on one server
- Nginx on another server

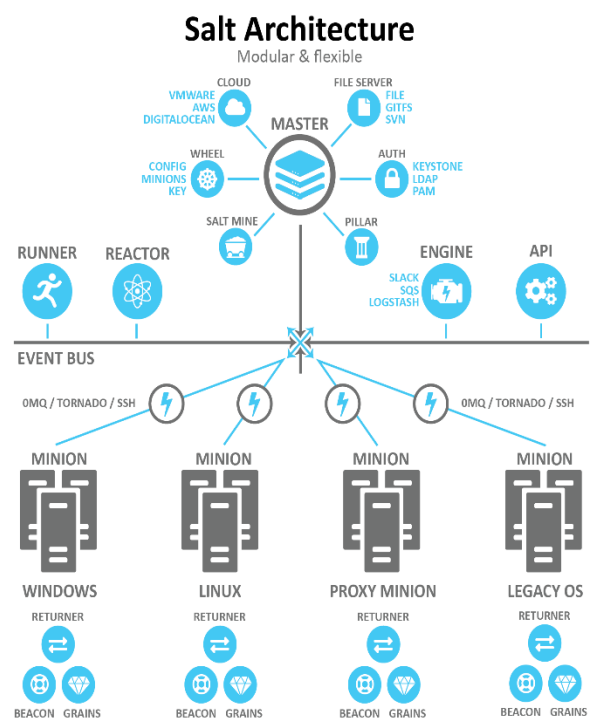
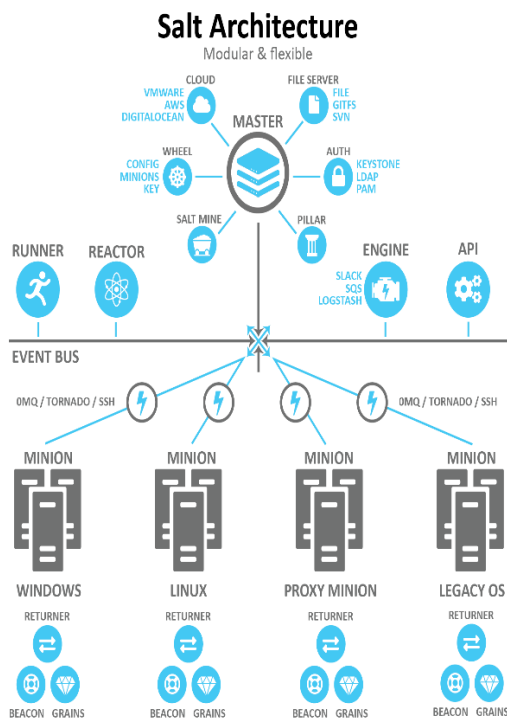
2. Objective:

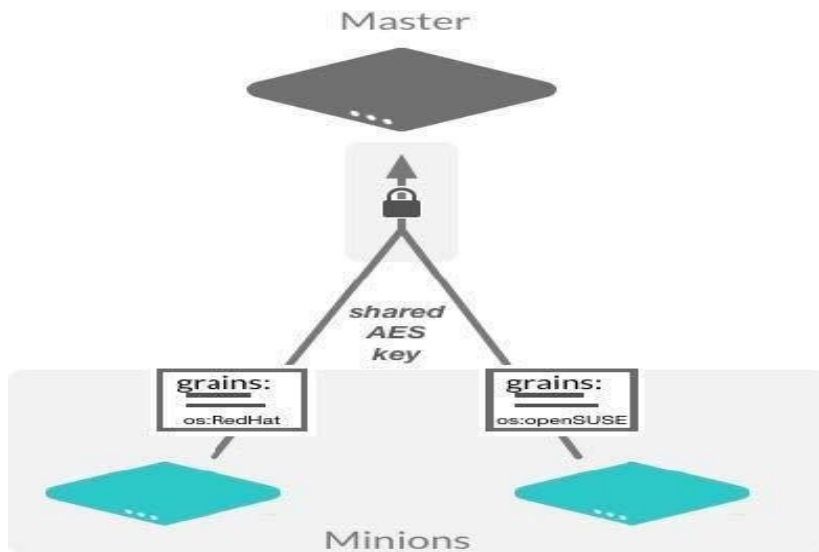
To implement configuration management using SaltStack's Master-Minion architecture and automate software installation.

3.Tools & Technologies Used:

- Amazon Web Services
- Ubuntu 22.04 EC2 Instances
- SaltStack
- Nginx
- OpenJDK 17

4. Architecture Diagram:





Components:

Component	Description
Salt Master	Central control server
Salt Minion	Managed EC2 instances
State File (.sls)	Configuration file
Grains	System information

5.Pre-Requisites:

- AWS Account
- 3 Ubuntu EC2 Instances
- Same Key Pair
- Ports Opened:

Server	Ports
Salt Master	22, 4505, 4506
Java Node	22
Nginx Node	22, 80

6.Procedure:

Step 1: Install Salt Master

Login to Master:

```
ssh -i key.pem ubuntu@<master-ip>
```

Install Salt Master:

```
sudo apt-get update
sudo apt-get install python3
```

```
wget -O bootstrap-salt.sh https://github.com/saltstack/salt-bootstrap/releases/latest/download/bootstrap-salt.sh
sudo sh bootstrap-salt.sh -M -N -P
```

Start service:

```
sudo systemctl start salt-master
sudo systemctl enable salt-master
sudo systemctl status salt-master
```

(Note:

Scenario:

1 Salt Master (Control server – EC2 Ubuntu)

2 Salt Minions

- Java Node
- Nginx Node

Salt works using a **Master–Minion architecture** where:

- **Master** → sends commands
- **Minions** → execute command)

Step 2: Install Salt Minion (On Java & Nginx Nodes)

On both nodes:

```
wget -O bootstrap-salt.sh https://github.com/saltstack/salt-bootstrap/releases/latest/download/bootstrap-salt.sh
```

```
sudo sh bootstrap-salt.sh -P
```

Configure minion:

```
sudo nano /etc/salt/minion
```

(note: This file connects the **Minion to the Master**)

Add:

```
master: <master private-ip>
```

(Note: This tells the **Minion where the Master server is located**)

Restart service:

```
sudo systemctl restart salt-minion
```

(Note: Restarting the service will:

1. Reload configuration
2. Connect to the Master
3. Send authentication request

The Minion now tries to **register with the Master.**)

Step 3: Accept Minion Keys (On Master)

(Note: When Minions connect, the Master receives authentication keys)

On Master:

```
sudo salt-key -L
```

Accept keys:

```
sudo salt-key -A
```

(Note: Now the Minions are **trusted by the Master**)

Step 4: Test Connection

```
sudo salt '*' test.ping
```

(Note: This “salt” command checks communication between Master and Minions

* → All Minions

test.ping → Test module)

Output:

True

Step 5: Create State File (on master)

Create directory:

```
sudo mkdir -p /srv/salt
```

(Note: Salt uses State Files (.sls) to define configurations.

A state file contains instructions for system configuration.

Default Directory:

/srv/salt/

Explanation:

Directory	Purpose
/srv/salt	Default Salt State directory
State files (.sls)	Configuration scripts

Create file:

```
sudo nano /srv/salt/install.sls
```

Add:

```
# Install Java on Java Node
```

```
install_java:
```

```
  pkg.installed:
```

```
    - name: openjdk-17-jdk
```

```
# Install Nginx on Nginx Node
```

```
install_nginx:
```

```
  pkg.installed:
```

```
    - name: nginx
```

```
start_nginx:
```

```
  service.running:
```

```
    - name: nginx
```

```
    - enable: True
```

Step 6: Apply State

Run from Master:

```
sudo salt '*' state.apply install
```

(Note:

* → all minions

state.apply → Apply state configuration

install → install.sls file)

7. Verification:

Verify Java Installation

On Java Node:
java -version

Verify Nginx Installation

Open browser:
http://<nginx-ip>
Nginx default page appears.

8. Output:

After execution, Salt displays:
Succeeded: X
Failed: 0

9.Result:

Successfully installed:

- Java on one EC2 instance
- Nginx on another EC2 instance

Using Salt Stack Master–Minion architecture.

10.Conclusion:

Salt Stack provides fast and scalable configuration management using a Master–Minion model and state files written in YAML.