

System Provisioning & Configuration Management.

Module # 03

Configuration Management Concepts & Tools

Copyright © 2023. All rights reserved. release 2.0.0

Module Learning Objectives

Upon successful conclusion of this module, you will be able to:

- Understand Configuration Management: Grasp principles and significance.
- Explore Key Components: Dive into CIs, Baselines, and Change Control.
- Appreciate Benefits: Recognize stability, security, and troubleshooting advantages.
- Introduction to DSC: Learn Desired State Configuration principles.
- Configuration Management Tools: Understand Ansible, Chef, Puppet, and SaltStack.
- Advanced Topics: Cover drift, CI/CD integration, and security considerations.
- Case Studies and Implementation: Analyze scenarios and apply hands-on labs.
- Explore Future Trends: Understand emerging technologies and industry best practices.



Module Topics

Understanding Configuration Management

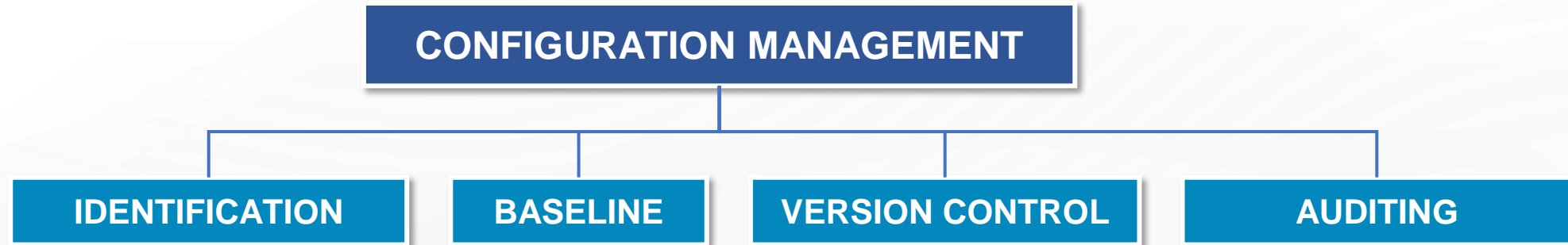
- Desired State Configuration (DSC)
- Introduction to Configuration Management Tools
 - ↳ Ansible, Chef, Puppet, SaltStack
- Advanced Topics in Configuration Management
- Case Studies and Practical Implementations



3.0.0 Understanding Configuration Management.

What is Configuration Management?

Systematic approach to identifying, controlling, and documenting system components



3.0.0 Importance of Configuration Management in IT

Lets understand the of configuration management in IT



Systematic Organization

Brings order to the complexity of IT systems



Navigating Rapid Evolution

Guides through the complexities of rapidly evolving IT

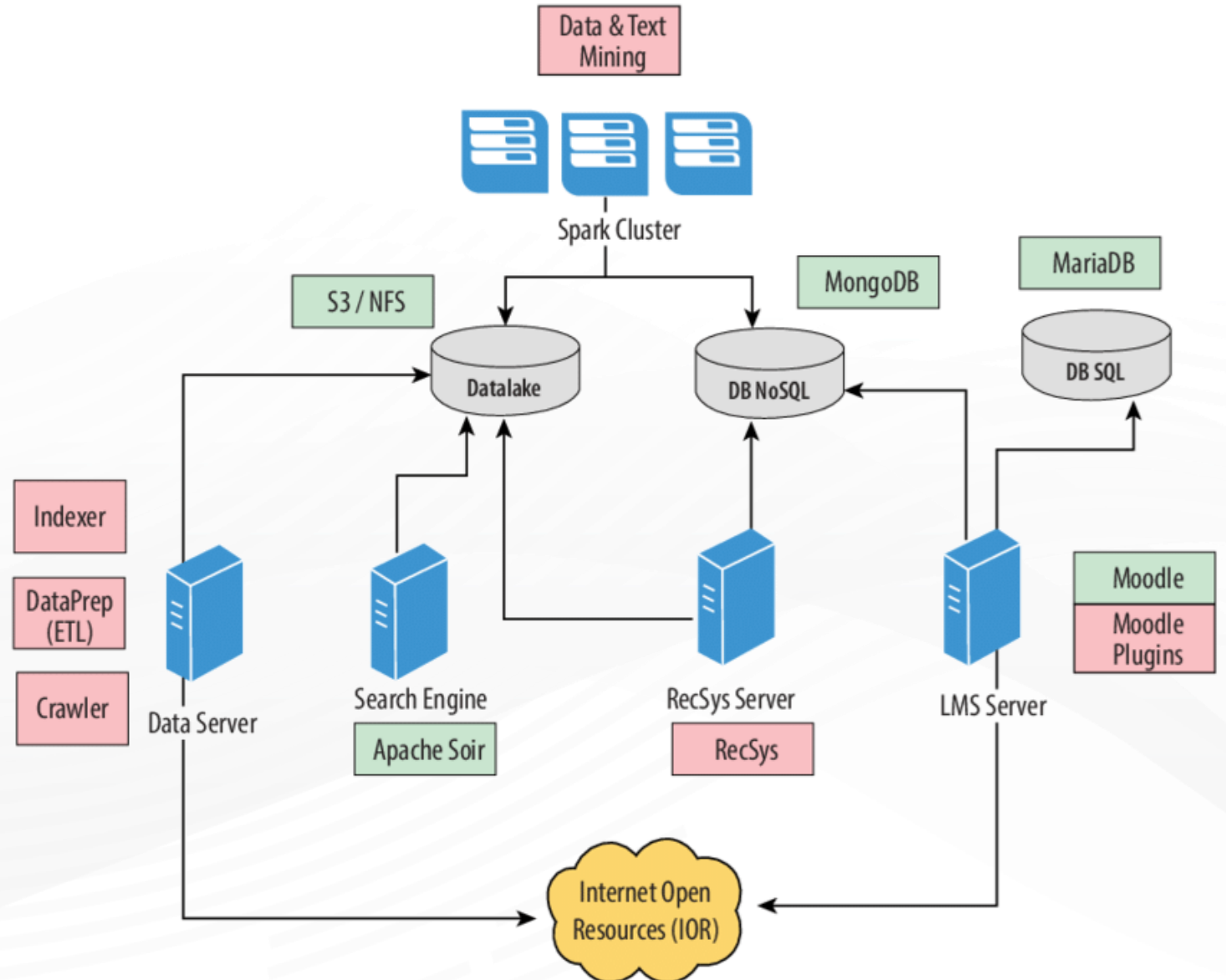


Consistency and Reliability

Ensures consistency and reliability in computing environments

3.0.1 Significance in System Administration and Infrastructure Deployment.

Configuration Management is integral to system administration and infrastructure deployment, ensuring consistency, reliability, and efficiency. By preventing unauthorized changes and facilitating uniformity, it plays a crucial role in maintaining system stability and support.



3.0.2 Challenges and Risks Without Proper Configuration Management.

Navigating Configuration Drift

Configuration drift, where systems gradually deviate from their intended state, leading to inconsistencies and unreliable environment.

Shielding Against Risk

Security vulnerabilities due to unmanaged configurations, exposing systems to unauthorized access and other threats.

Unraveling Technical Quandaries

Troubleshooting complexities caused by lack of configuration management, leading to difficulties in identifying and resolving issues.

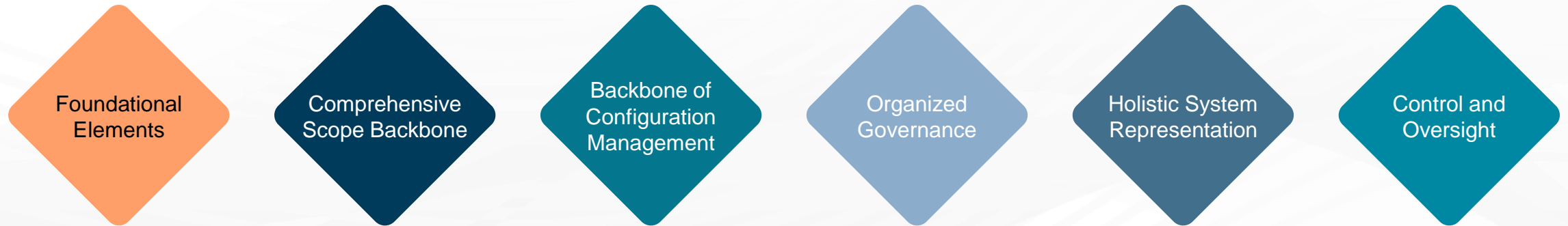
Navigating Regulatory Waters

Compliance issues, as the absence of proper configuration management may result in failure to meet regulatory requirements.

3.0.3 Key Components of Configuration Management.

Configuration Items (CIs) are the fundamental building blocks of a system, encompassing hardware, software, documentation, and more. They form the backbone of Configuration Management, aiding in the systematic organization and control of IT components.

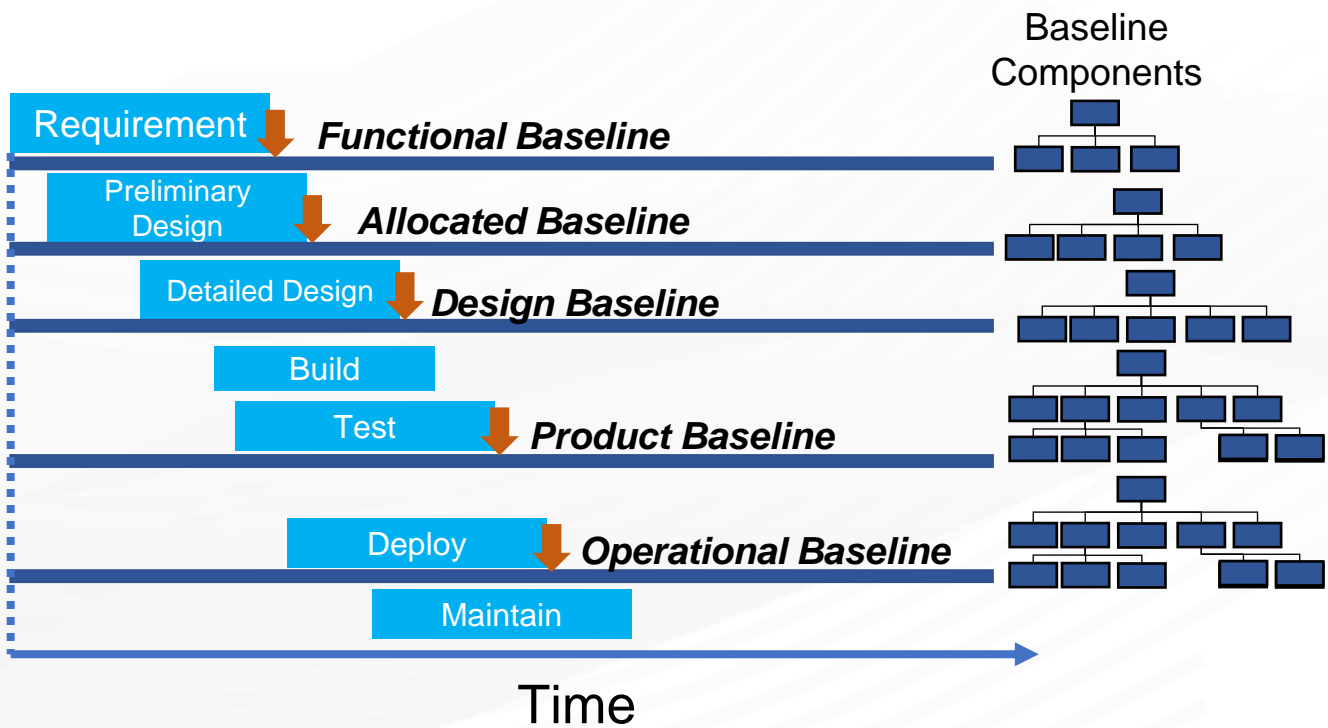
Configuration Items (CIs) in a System.



3.0.3 3.0.3 Key Components of Configuration Management Contd..

Configuration Baselines.

Configuration Baselines are predefined sets of specifications and standards that define the desired state of a system at a specific point in time. They serve as reference points for configuration control and change management.



3.0.4 Change Control and Versioning.

01

Importance

Crucial processes in Configuration Management, carefully managing modifications to configurations, providing a structured approach to assess the impact of changes.

02

Function of Change Control

Minimizing potential disruptions by carefully managing modifications to configurations, assessing the impact of changes.

03

Role of Versioning

Track system evolution over time, provide a historical record of changes.

3.0.5 Benefits of Effective Configuration Management | Improved Stability and Reliability.

Improved Stability and Reliability.

- Effective Configuration Management maintains a consistent and documented state, preventing unauthorized changes.
- It reduces the risk of disruptions in the IT environment, leading to quicker system recovery.
- The IT environment experiences improved stability and reliability due to these factors.

3.0.6 Benefits of Effective Configuration Management | Enhanced Security.

Enhanced Security.

- Prevents unauthorized access and minimizes security vulnerabilities.

- Establishes a robust security posture in IT environments.

- Minimizes the risk of security breaches.

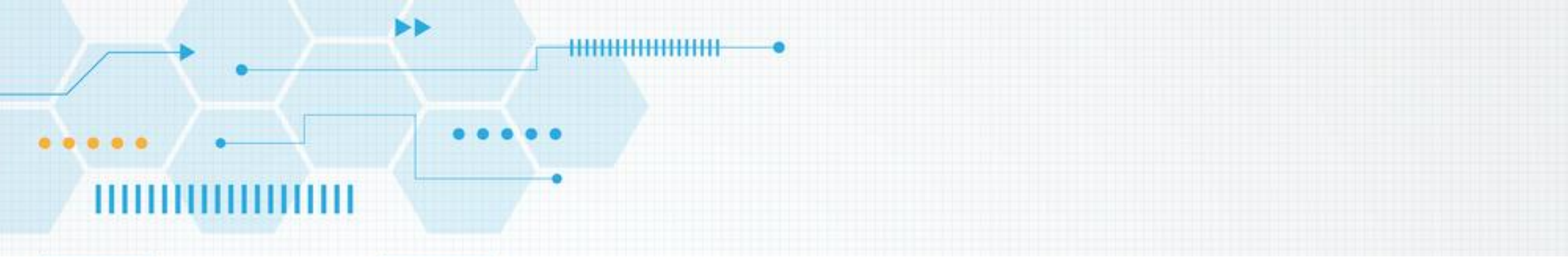
3.0.7 Benefits of Effective Configuration Management | Efficient Troubleshooting and Problem Resolution.

Efficient Troubleshooting and Problem Resolution.

- Well-documented configurations streamline the identification of issues.

- Reducing downtime and enhancing overall IT support.

- Efficient troubleshooting and problem resolution are facilitated by Configuration Management.

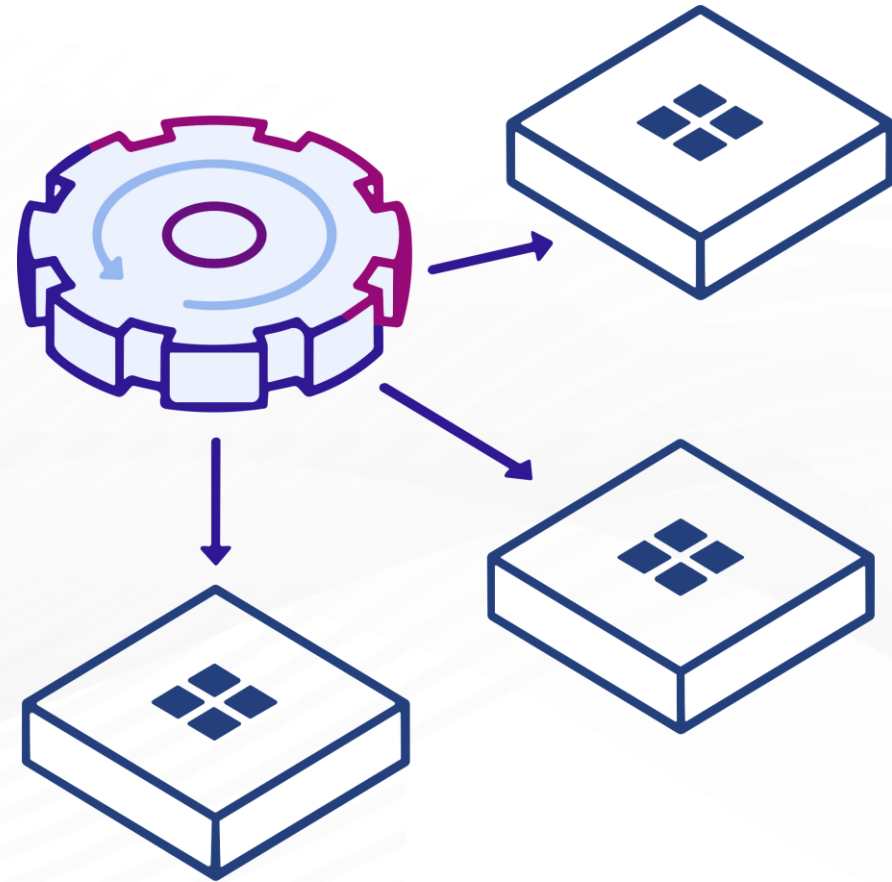


3.1.0 Desired State Configuration (DSC)



3.1.1 Introduction to DSC

Desired State Configuration (DSC) is a powerful configuration management framework in Windows environments. It operates on the principles of idempotence and declarative configuration.



3.1.2 Historical Context and Evolution

Emergence of DSC

DSC emerged as part of PowerShell 4.0 in 2013, addressing the initial challenges of configuration management in IT.

Evolution of DSC

DSC has evolved over the years to address the growing complexities of modern IT, becoming an integral part of automation and infrastructure as code.

Integration with PowerShell

Its integration with PowerShell provides a robust platform for automation, enabling administrators to leverage the full power of PowerShell scripting to define and enforce configurations.

Role in IaC

This synergy also positions DSC at the forefront of the Infrastructure as Code (IaC) movement, where infrastructure configurations are managed in a code-like manner, promoting efficiency and consistency.

3.1.3 Idempotence in Desired State Configuration.

What is Idempotence?

In DSC, idempotence ensures that no matter how many times you apply a configuration, the end result is the same. This reliability is a game-changer for maintaining stability and predictability in system configurations.



3.1.4 Ensuring Idempotence in Configuration Scripts.

DSC achieves idempotence by enforcing a 'desired state' through configuration scripts written in PowerShell. These scripts declare the intended configuration, and DSC handles the details of achieving and maintaining that state.



DSC ensures idempotence by enforcing a 'desired state.'



Configuration scripts in PowerShell declare the intended configuration.



DSC handles the details of achieving and maintaining that state.

3.1.4 Ensuring Idempotence in Configuration Scripts Contd..

Declarative vs. Imperative Models

Declarative

- The focus is on the desired end state of the system.
- Developers specify what outcomes they want, not how to achieve them.
- Well-suited for DSC as it aligns with the 'desired state' principle.

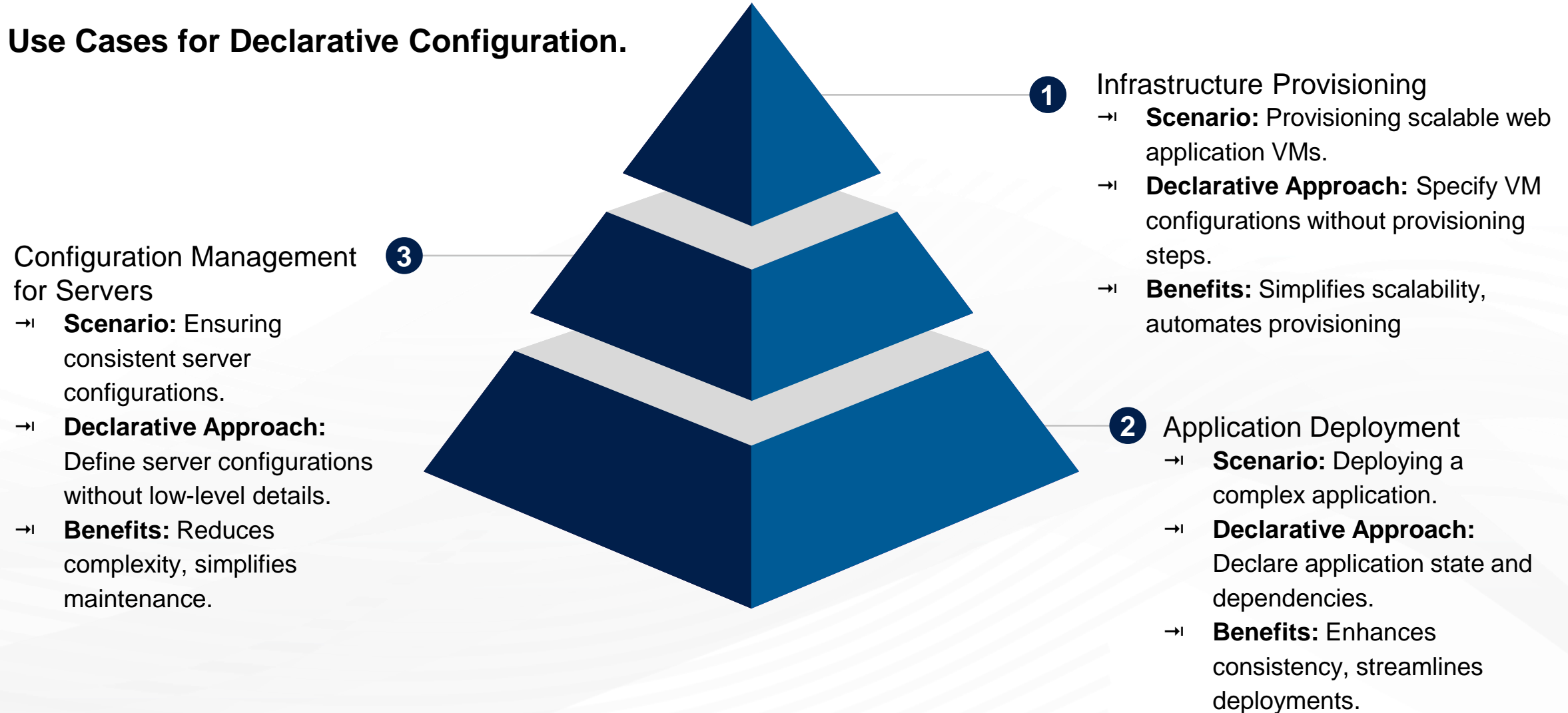
Imperative Models

- The emphasis is on the step-by-step process to achieve the state.
- Developers define the exact steps and commands to reach the state.
- Common in traditional configuration management approaches.



3.1.4 Ensuring Idempotence in Configuration Scripts Contd..

Use Cases for Declarative Configuration.

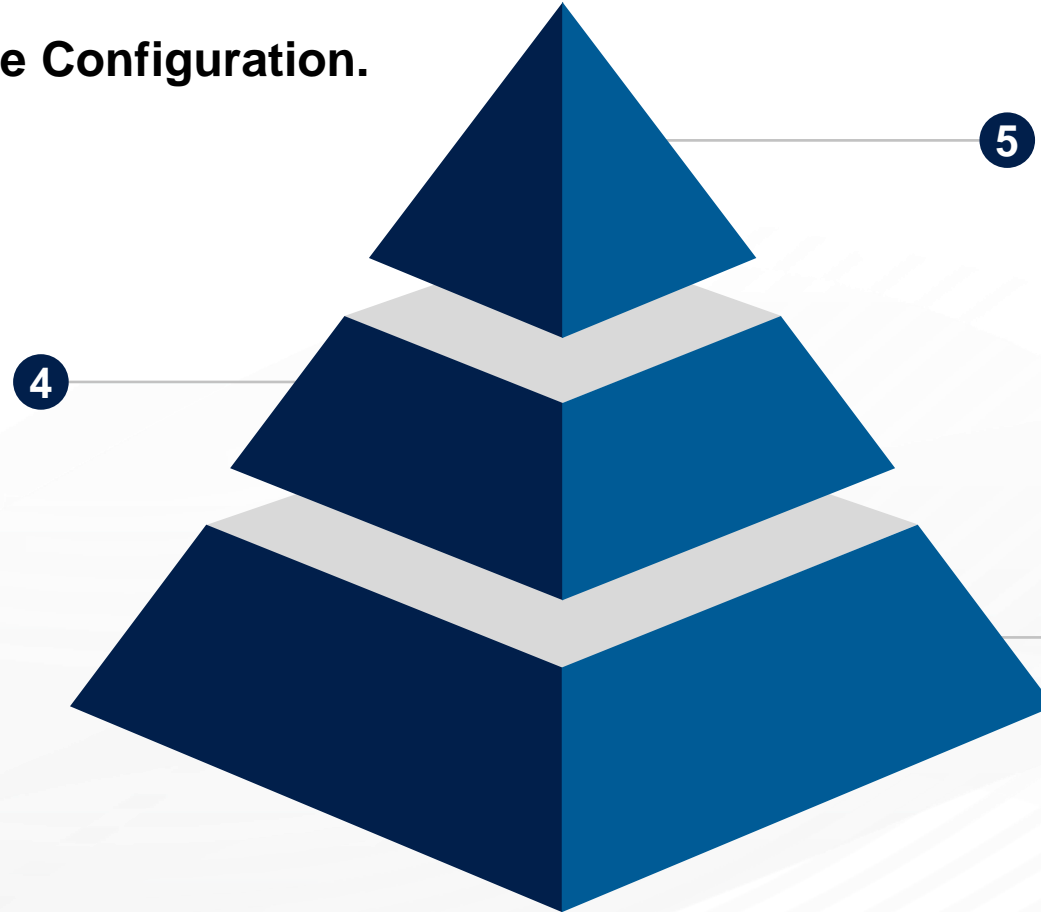


3.1.4 Ensuring Idempotence in Configuration Scripts Contd..

Use Cases for Declarative Configuration.

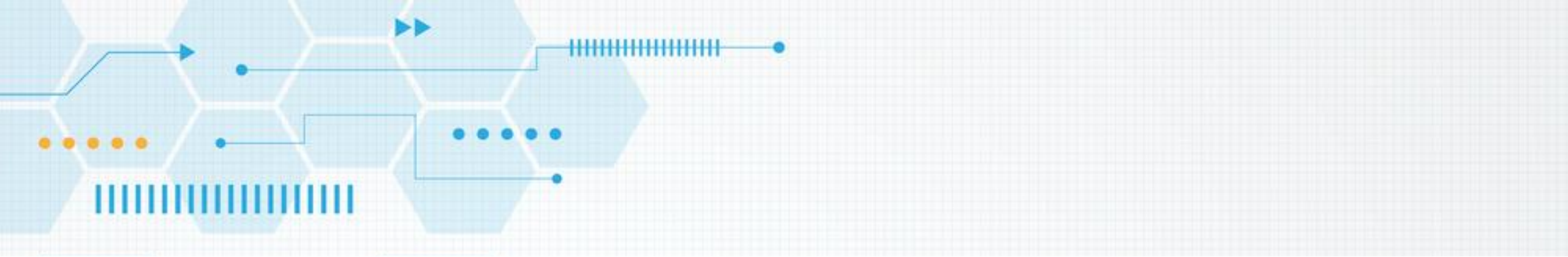
Network Infrastructure Configuration

- **Scenario:** Configuring data center network devices.
- **Declarative Approach:** Declare network topology and settings.
- **Benefits:** Streamlines network management.



- 5 Cloud Resource Management
 - **Scenario:** Managing cloud resources.
 - **Declarative Approach:** Specify resource configurations using templates.
 - **Benefits:** Facilitates agility, abstracts provisioning complexities.

- 6 Configuration Drift Remediation
 - **Scenario:** Addressing configuration drift.
 - **Declarative Approach:** Declare ideal configurations and enforce regularly.
 - **Benefits:** Efficiently corrects drift, ensures consistency.



3.2.0 Introduction to Configuration Management Tools.



3.2.1 Overview of Configuration Management Tools.

Configuration Management Tools serve as the backbone of modern IT infrastructure management, providing a systematic and automated approach. Now, let's delve deeper into the various types of Configuration Management Tools that cater to different needs and preferences within the IT community.

CONFIGURATION MANAGEMENT TOOLS



Terraform

'SALTSTACK



puppet



git



CHEF



ANSIBLE



docker

Role in Automating Infrastructure.

Automation of Configuration.

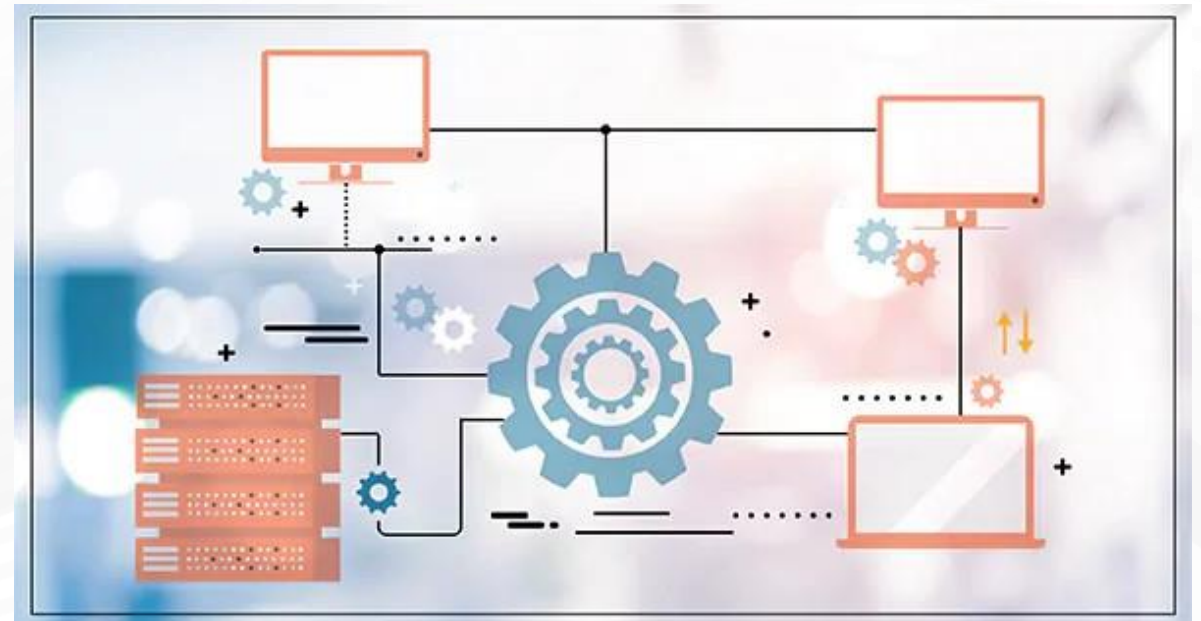
Ensuring Consistency

Reducing Manual Errors

Systematic Approach to Changes

Scalability and Management of Complexity

Efficiency in Operations



3.2.3 Types of Configuration Management Tools.

1

Agent-Based Configuration Management Tools

→ Agents communicate with a central server to receive and apply configurations. Examples include Puppet and Chef, which use a master-agent architecture.

2

Agentless Configuration Management Tools

→ Operate without requiring agents on managed systems. Ansible is a notable example

3

Scripting-Based Configuration Management Tools:

→ Rely on scripting languages to define and execute configuration tasks. Examples include Shell scripts and PowerShell scripts

4

Event-Driven Configuration Management Tools

→ React to events and changes in the infrastructure to trigger automated responses. Example is SaltStack.

3.2.4 Configuration Management Tool Comparison.

Feature	Ansible	Chef	Puppet	SaltStack
Architecture	Agentless, masterless architecture	Master-server and agent architecture	Master-server and agent architecture	Master-minion architecture
Language	YAML	Ruby DSL	Puppet DSL	YAML and Jinja templating
Learning Curve	Low	Moderate	Moderate	Moderate
Integration Capabilities	Excellent integration with various tools/platforms	Integrates well with various tools/platforms, Extensive ecosystem	Extensive integrations with tools/platforms, Rich ecosystem	Integrates well with various tools/platforms, Active community
Use Cases	General automation	Infrastructure automation	Infrastructure automation	Event-driven automation



ANSIBLE



3.2.5 Overview of Ansible Architecture

Ansible is an open-source automation tool that simplifies configuration management, application deployment, and task automation. It follows a client-server architecture and operates over SSH, making it agentless. The primary components of Ansible's architecture include:



3.2.6 Control Node.

- The control node is where Ansible is installed and where playbooks are executed.
- It contains the inventory file, which specifies the managed nodes and their connection details.
- The control node communicates with managed nodes over SSH, which must be pre configured for passwordless access

Example:

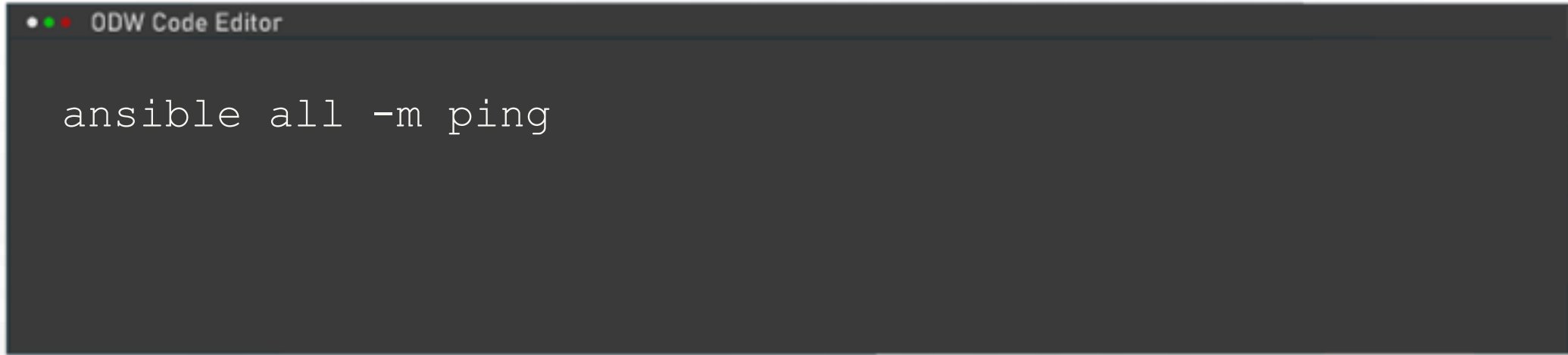


```
ansible-playbook -i inventory.ini my_playbook.yml
```

3.2.7 Managed Nodes.

- These are the servers or devices that Ansible manages. Managed nodes must have Python and SSH installed.
- Ansible communicates with managed nodes using SSH to execute tasks defined in playbooks.

Example:

A screenshot of a terminal window with a dark background. The title bar at the top left shows three colored dots (red, yellow, green) followed by the text 'ODW Code Editor'. The main area of the terminal displays the command 'ansible all -m ping' in a light-colored monospaced font.

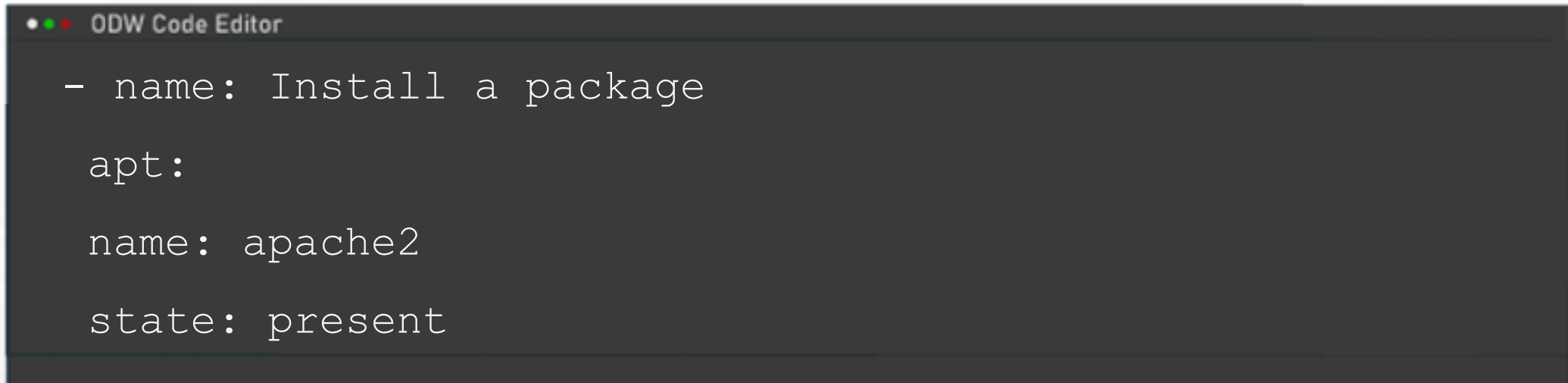
```
ODW Code Editor

ansible all -m ping
```

3.2.8 Modules.

- Modules are units of code responsible for carrying out specific tasks on managed nodes.
- They are executed on the managed nodes and can be written in any language that returns JSON.

Example:

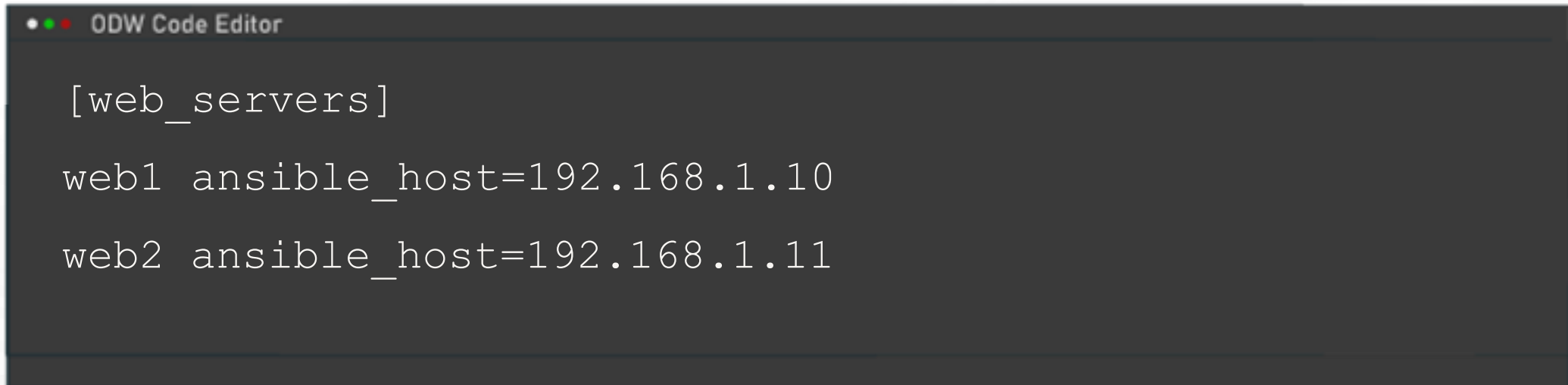


```
• ● ● ODW Code Editor
- name: Install a package
  apt:
    name: apache2
    state: present
```

3.2.9 Inventory.

- The inventory file is where details about managed nodes are stored, including IP addresses, hostnames, and groupings.
- It can be static or dynamic, allowing for flexibility in defining and managing hosts.

Example:

A screenshot of a code editor window titled "ODW Code Editor". The window has a dark background and displays the following text:

```
[web_servers]
web1 ansible_host=192.168.1.10
web2 ansible_host=192.168.1.11
```


3.2.10 Playbooks.

- Playbooks are written in YAML and describe a set of tasks to be executed on managed nodes.
- They provide a higher-level abstraction, allowing users to define configurations and sequences of tasks.

Example:

```
ODW Code Editor

- name: Ensure Apache is installed and running
  hosts: web_servers
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
    - name: Start Apache
      service:
        name: apache2
        state: started
```

3.2.11 API (Application Programming Interface).

- Ansible provides an API for integration with external tools, enabling automation workflows.
- The API allows for programmatic control of Ansible, facilitating interactions from other applications.

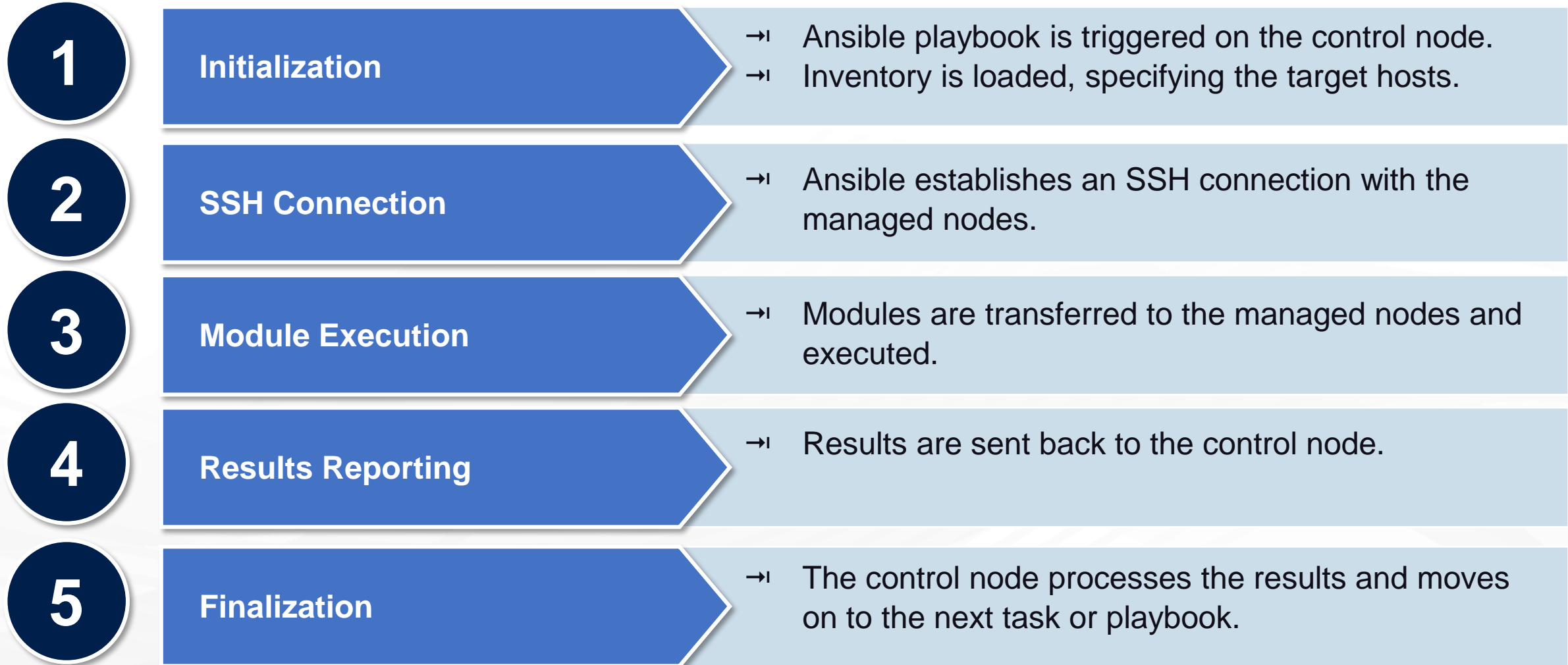
Example:

```
ODW Code Editor

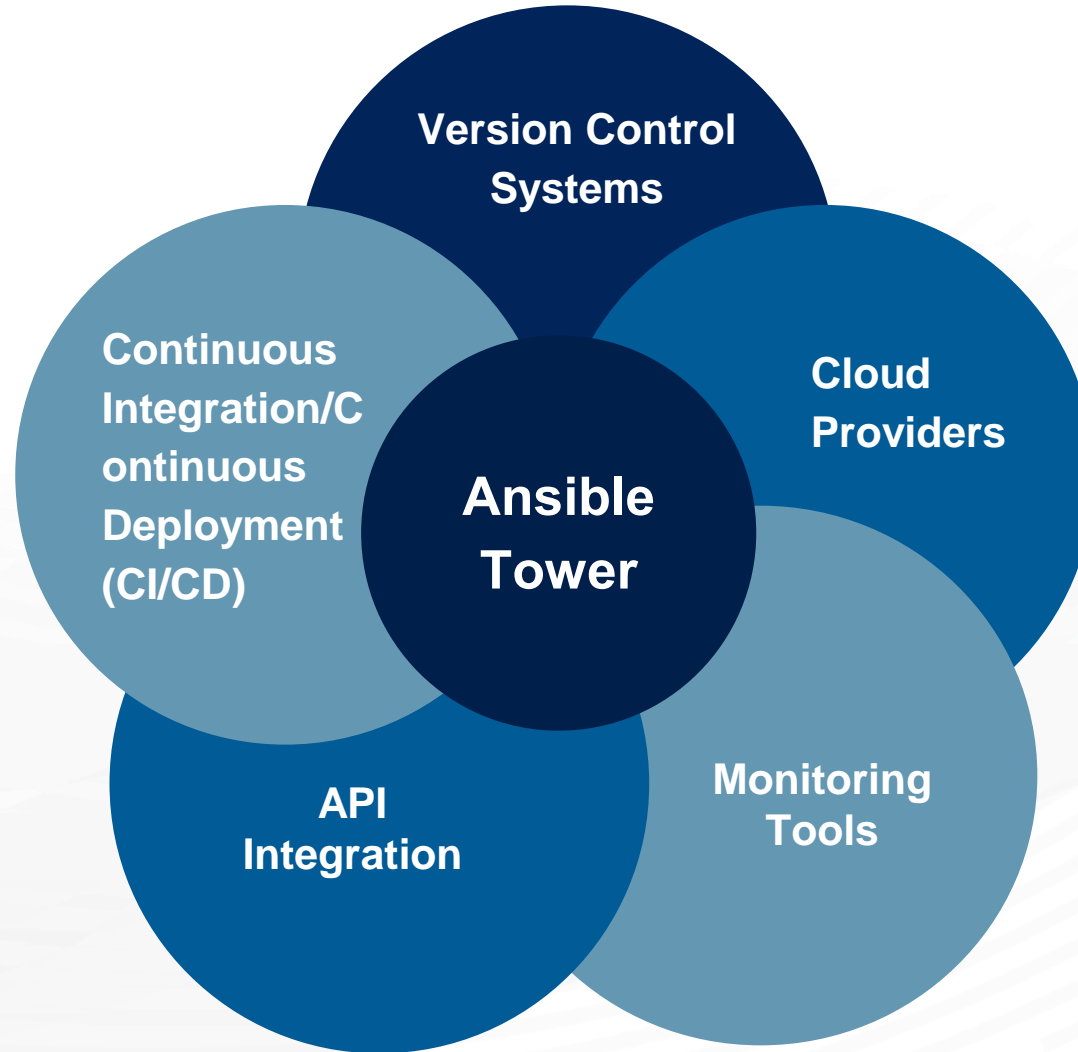
import ansible_runner

# Run Ansible playbook programmatically
ansible_runner.run(private_data_dir='/path/to/playbook'
, playbook='my_playbook.yml')
```

3.2.12 Execution Flow.



3.2.13 Integration with Other Tools and Platforms.





CHEF

3.2.14 Chef Infra

→ Chef Infra is a powerful configuration management tool that automates the deployment and management of infrastructure. Key components include Cookbooks, Recipes and Nodes.

Example: Cookbook: 'webserver'

```
• • • ODW Code Editor

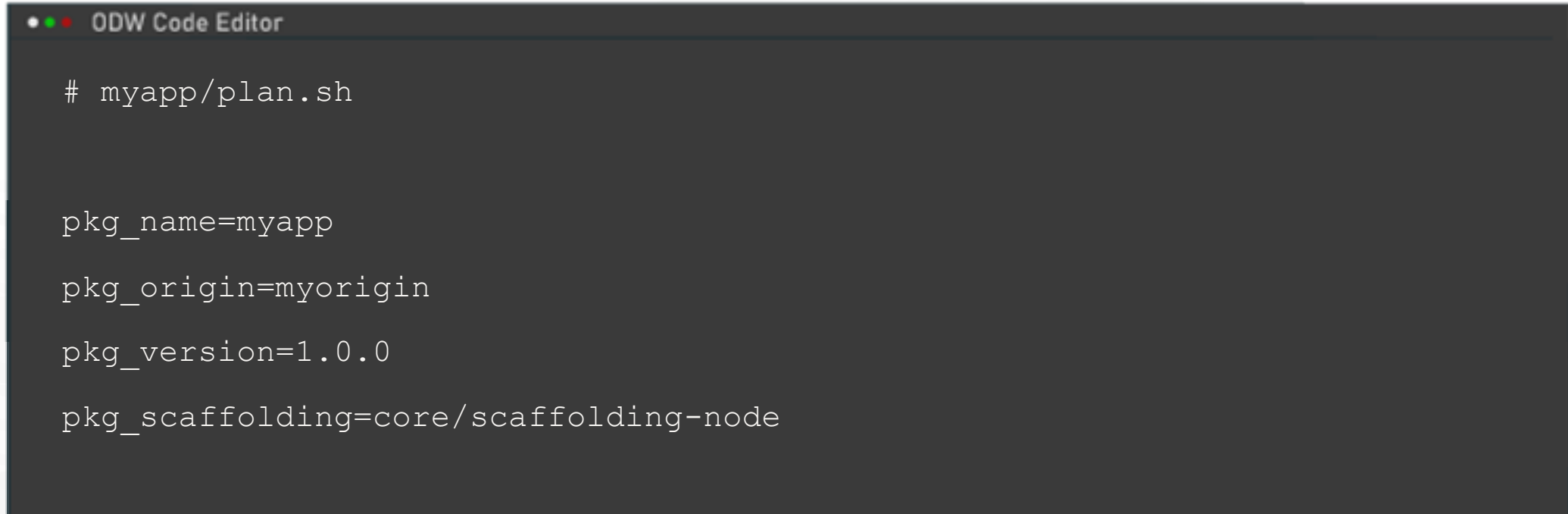
# webserver/recipes/default.rb
package 'apache2'
service 'apache2' do
  action [:enable, :start]
end
template '/var/www/html/index.html' do
  source 'index.html.erb'
  variables(
    title: 'Welcome to My Website',
    message: 'This web server is managed by Chef Infra!'
  )
end
```

3.2.15 Chef Habitat.

Chef Habitat

is designed for application-centric automation. Key features include Application Packaging, Service Groups, Supervisor.

Example: Plan: 'myapp'

A screenshot of a code editor window titled "ODW Code Editor". The editor displays a Chef Habitat plan script for "myapp". The script is written in a light-colored font on a dark background. It starts with a comment line "# myapp/plan.sh" followed by four lines of configuration: "pkg_name=myapp", "pkg_origin=myorigin", "pkg_version=1.0.0", and "pkg_scaffolding=core/scaffolding-node".

```
• ● ● ODW Code Editor

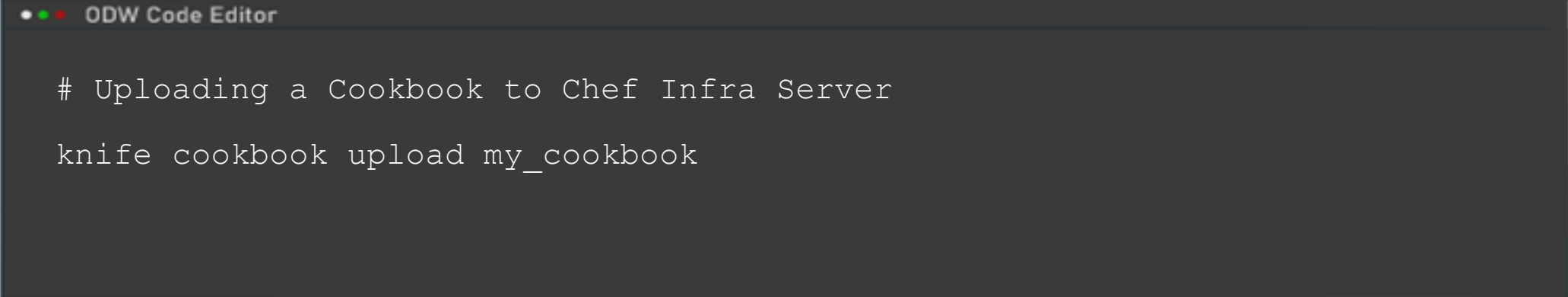
# myapp/plan.sh

pkg_name=myapp
pkg_origin=myorigin
pkg_version=1.0.0
pkg_scaffolding=core/scaffolding-node
```


3.2.16 Chef Workstation.

Chef Workstation is the development environment for Chef users. It provides a set of tools for authoring, testing, and maintaining Chef Infra configurations and Chef Habitat packages. Key components of Chef Workstation include: **ChefDK, Knife**

Knife Command:



```
• • • ODW Code Editor

# Uploading a Cookbook to Chef Infra Server

knife cookbook upload my_cookbook
```

The knife command is used to interact with Chef Infra Server. In this example, it uploads a cookbook named `'my_cookbook'` to the server

3.2.17 Chef Automate.

Chef Automate is a platform that provides visibility into the entire infrastructure automation **workflow**. It acts as a central hub for managing and monitoring Chef Infra and Chef Habitat environments. Key features of Chef Automate include: **Compliance Automation, Visibility and Reporting, Workflow Automation.**

Compliance Profile: 'baseline'

```
ODW Code Editor

# baseline.rb

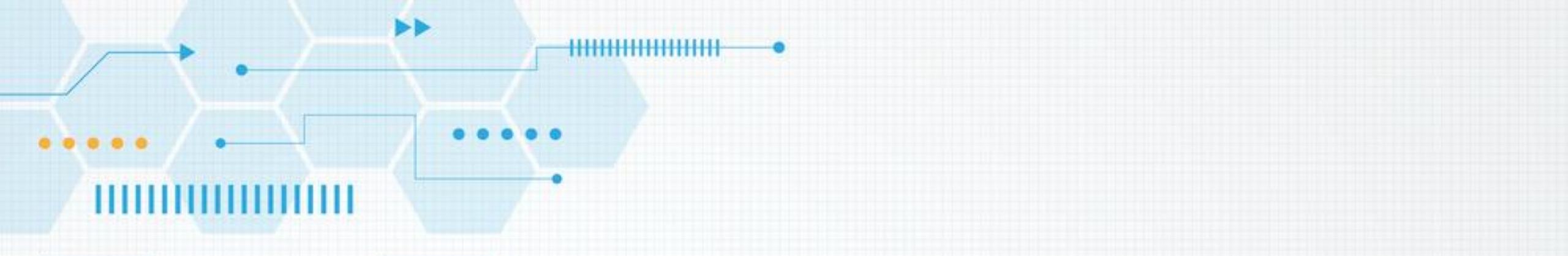
control 'os-01' do

  impact 1.0

  title 'Ensure password complexity is enforced'

  describe command('security_policy') do

    its('PasswordComplexity') { should eq 1 }
  end
end
```



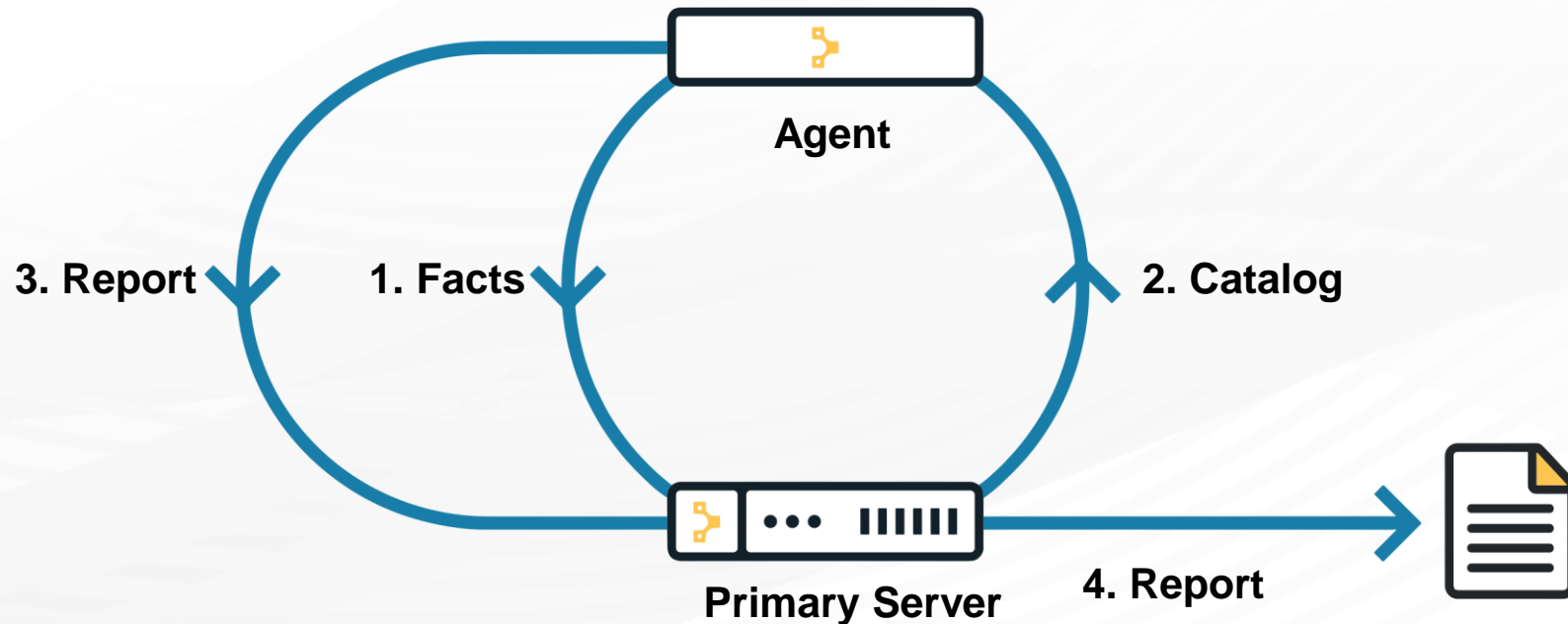
puppet



3.2.18 Puppet.

Introduction to Puppet

Puppet is a powerful open-source configuration management tool designed to automate the provisioning and management of IT infrastructure. It enables system administrators to define and enforce the desired state of their infrastructure, ensuring consistency and repeatability.

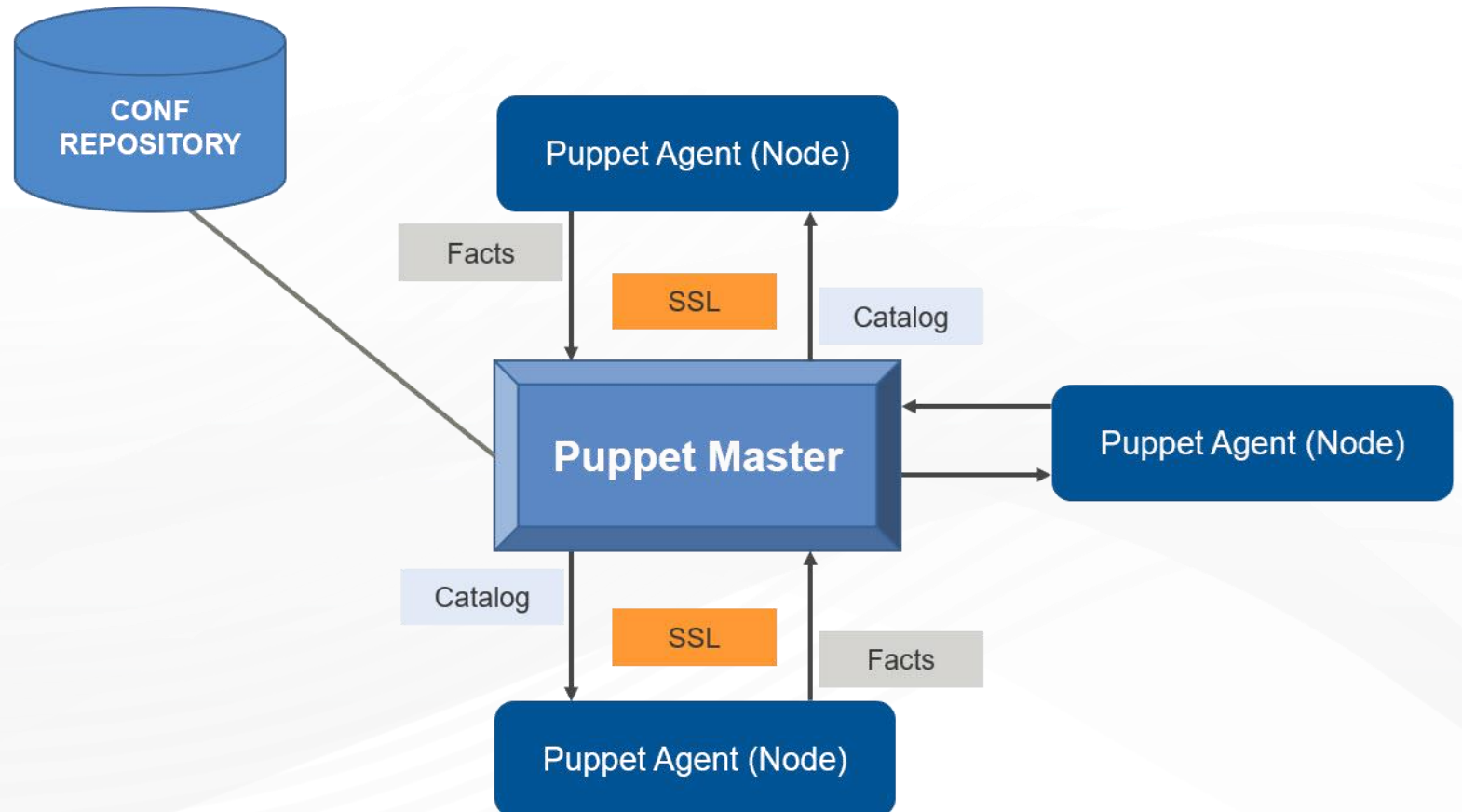


3.2.18 Puppet.

Puppet Architecture

Puppet follows a client-server model, comprising two main components: the Puppet master and Puppet agents.

Let's read more in the next slide.



3.2.19 Puppet Master



Catalog Compiler

Compiles catalogs, representations of the desired system state, using Puppet DSL



Certificate Authority (CA)

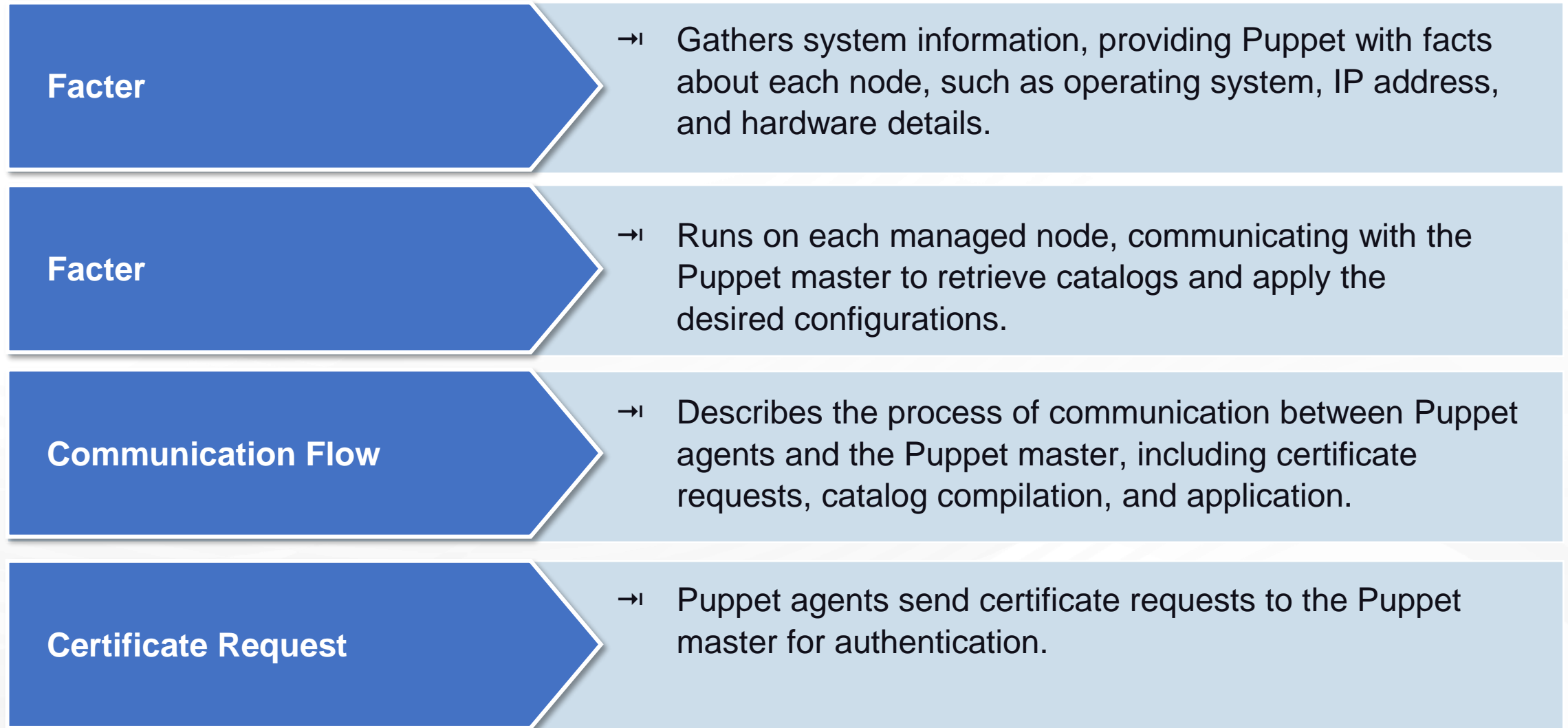
Responsible for authenticating and authorizing Puppet agents before they can



PuppetDB

Stores data generated by Puppet, providing a centralized repository for information about nodes, facts, and resources.

3.2.20 Puppet Agents & Communication Flow.



3.2.21 Key Terminology



Node

Any device or server managed by Puppet.



Manifest

Puppet code specifying the desired state of a node, including configurations and dependencies.



Resource

A unit of configuration managed by Puppet, representing a component of the system (e.g., a file, package, or service).



Module:

A collection of manifests and associated files organized to solve specific problems or manage particular resources.



Catalog

A compiled representation of the desired system state generated by the Puppet master for each managed node



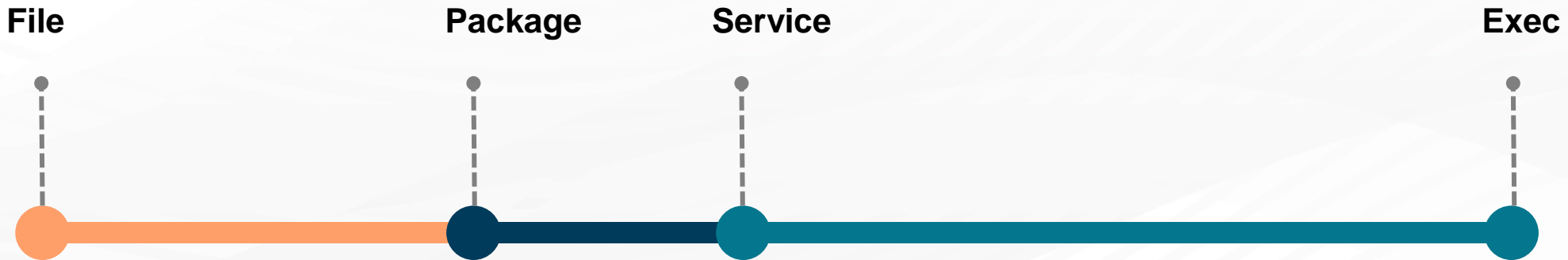
Puppet Forge

A centralized repository for Puppet modules, facilitating the sharing and discovery of pre-built configurations.

3.2.22 Puppet Manifests

Manifests are written in Puppet DSL and define the desired state of a node. They consist of resources, relationships, and conditional statements, providing a clear and declarative approach to system configuration.

Syntax



3.2.23 Puppet Forge.

Puppet Forge serves as a repository for Puppet modules, providing a centralized location for sharing and discovering pre-built configurations.

Administrators can leverage Puppet Forge to accelerate infrastructure management by incorporating tested and community-supported modules into their environments.

Module Discovery:
Users can explore and search for modules based on their requirements, saving time and effort in configuration development.

Versioning: Puppet Forge supports versioning, allowing users to specify module versions to ensure compatibility and stability.

3.2.24 Puppet Ecosystem

Hiera

A key-value data lookup tool that separates data from code, facilitating the management of configuration data.



Bolt

Puppet's task and automation tool, enabling the execution of ad-hoc tasks and scripts on remote nodes.



Puppet Development Kit (PDK):

A development environment that streamlines module creation, testing, and documentation.



Continuous Integration (CI) Integration

Puppet integrates seamlessly with CI tools like Jenkins and Travis CI, automating the testing and deployment of Puppet code





SALTSTACK

3.2.25 Salt Master and Minion Architecture.

SaltStack, often referred to simply as Salt, is a powerful open-source configuration management and automation tool.

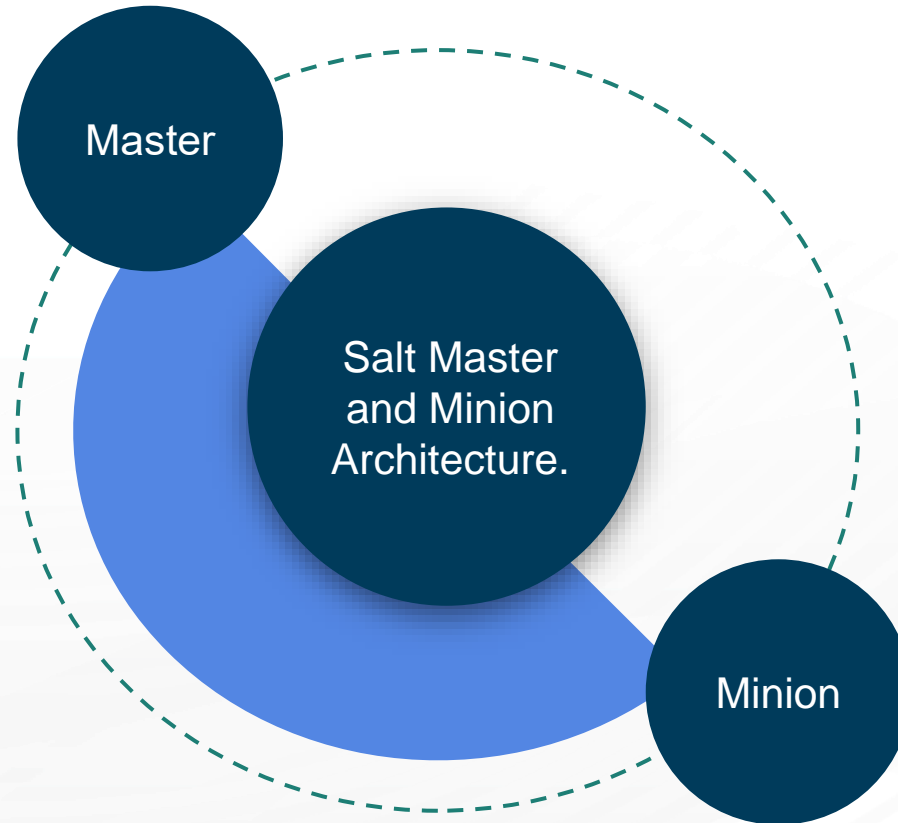
Master

- Manages the Minions
- Sends commands to Minions
- Stores key information for Minion authentication

Minion

- A client that connects to the Master
- Executes the commands and reports back to the Master
- Each Minion has a unique ID

3.2.25 Salt Master and Minion Architecture.



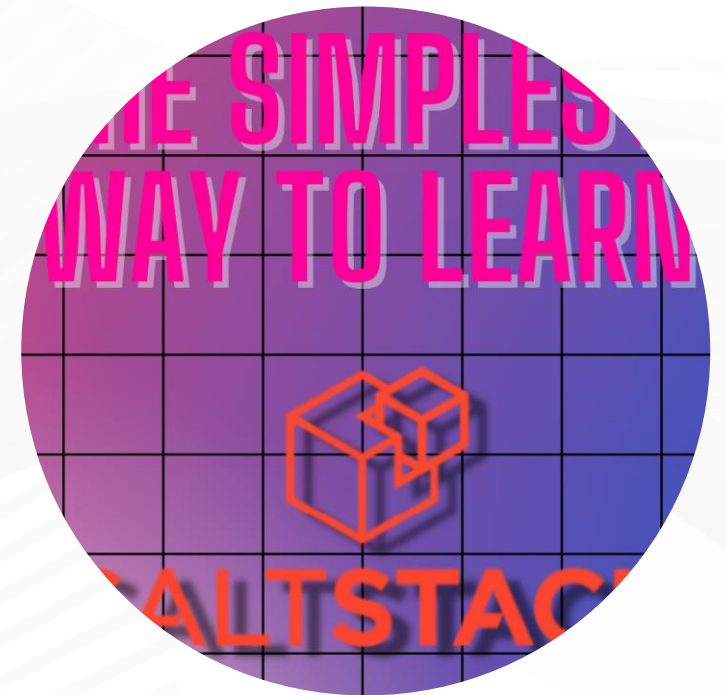
SaltStack, often referred to simply as Salt, is a powerful open-source configuration management and automation tool.

3.2.26 Understanding States and Formulas in SaltStack.

States are the desired configurations for systems, such as packages installed or services running.

Formulas are the files that define the states and are written in YAML format.

The SaltStack formula is a collection of files, including states, pillar data, and other necessary files.



3.2.27 SaltStack's Dynamic Automation Trio: Event System, Reactor System, and Orchestration

Event System

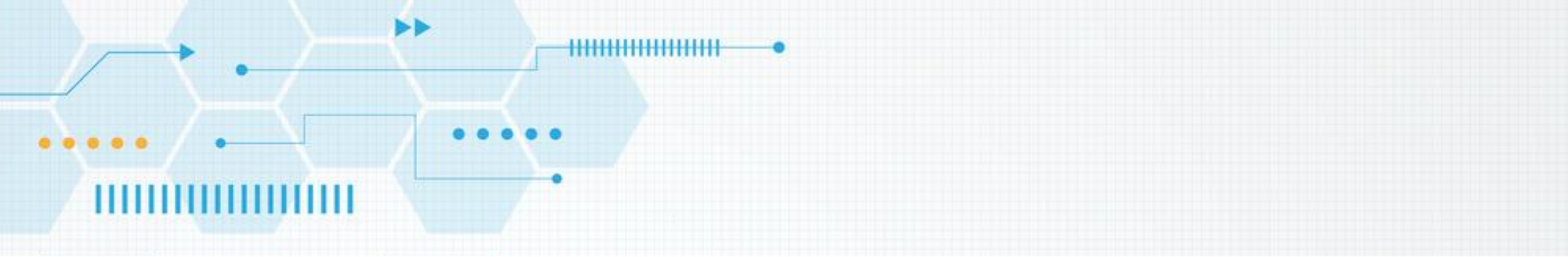
- Real-Time Responsiveness
- Broadcasted Intelligence

Reactor System:

- Automated Event Responses
- Configurable Reactions
- Real-Time Infrastructure State Management

Orchestration

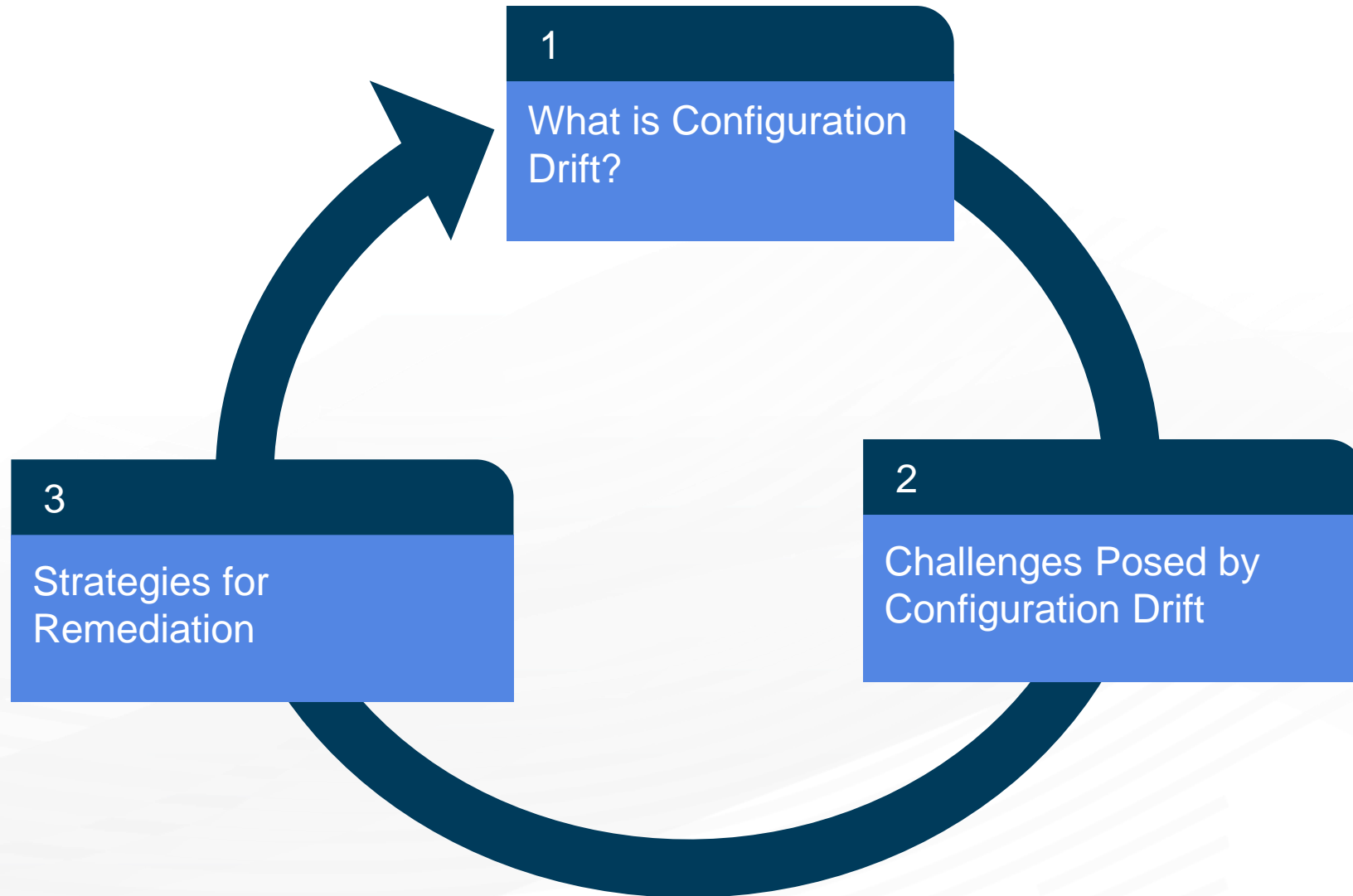
- Workflow Customization
- Cross-Minion Coordination
- Scale and Complexity Management



3.3.0 Advanced Topics in Configuration Management



3.3.1 Configuration Drift and Remediation.



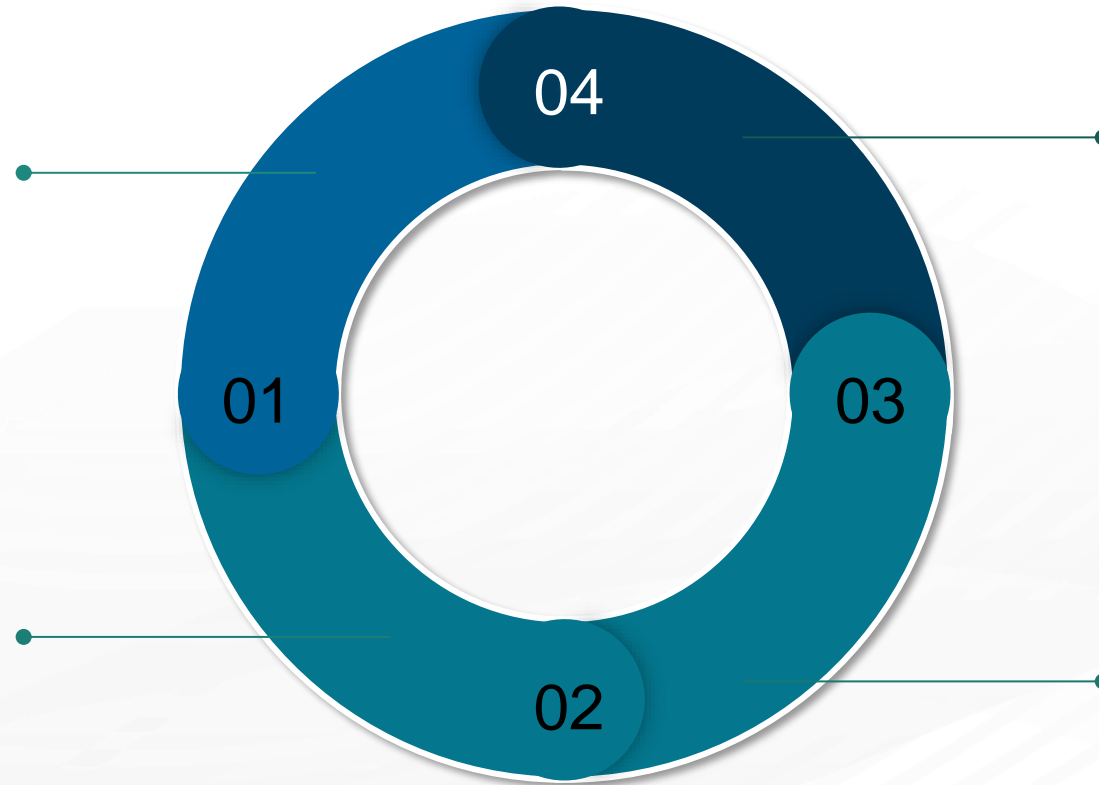
3.3.2 Identifying and Managing Configuration Drift.

Forms of Drift

Configuration drift can manifest in different forms, including changes in system settings, file configurations, or installed software versions.

Tools for Detection

Comprehensive monitoring and auditing tools are required to compare the current state of a system with its predefined configuration baseline.



Real-time Monitoring

Real-time monitoring is crucial for promptly identifying configuration drift. Automated alerts and notifications can further expedite the response process.

Example Tools

Examples of tools for detecting configuration drift include Ansible, Puppet, Chef, Tripwire, and AIDE.

3.3.3 Strategies for Configuration Remediation.



Automation plays a pivotal role in addressing configuration drift. It can bring systems back to their intended state by applying predefined configuration settings.



Implementing rollback mechanisms is a proactive strategy to mitigate the impact of configuration drift. Organizations can revert to a known-good state in case of unexpected changes.



Adopting a continuous compliance monitoring approach involves regularly assessing systems against industry standards and regulatory requirements.



Establishing robust change management policies is essential for preventing unauthorized configuration changes.



3.3.4 Integration with Continuous Integration/Continuous Deployment (CI/CD) Pipelines

Seamless Integration of Configuration Management in CI/CD

- The seamless integration of configuration management in CI/CD requires a systematic approach to handle configurations, dependencies, and environmental variables.
- It plays a pivotal role in maintaining consistency across various environments, reducing errors, and enhancing overall deployment efficiency.
- Configuration management is essential for ensuring that code is deployed without unexpected issues in a CI/CD pipeline.

3.3.5 Key Components of Seamless Integration.

1

Infrastructure as Code (IaC)

2

Version Control Systems (VCS)

3

Automated Configuration Deployment

4

Continuous deployment of configuration:

5

Configuration Validation

5

Environment-specific Configurations

3.3.5 Best Practices.

Immutable Infrastructure



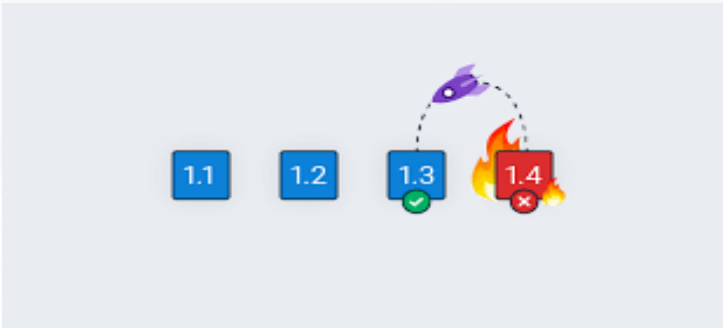
Continuous Monitoring



Documentation



Rollback Mechanism



3.3.6 Automated Testing and Validation in CI/CD Pipelines.

Automated testing is a critical component of CI/CD pipelines, ensuring that code changes are systematically validated before deployment.

The process of automated testing and validation helps maintain the integrity of the software by identifying and eliminating critical bugs and ensuring high quality.

In the context of CI/CD, automated testing and validation are essential for achieving the goals of continuous integration, continuous delivery, and frequent, reliable releases.

3.3.7 Types of Automated Testing in CI/CD.

Unit Testing

Focused on individual units of code, unit tests validate that each component functions as intended. Automated unit testing is executed during the early stages of the CI/CD pipeline, providing rapid feedback to developers.

Integration Testing

This type of testing evaluates the interactions between different components or modules. In a CI/CD pipeline, automated integration tests verify that the integrated code functions as a cohesive unit.

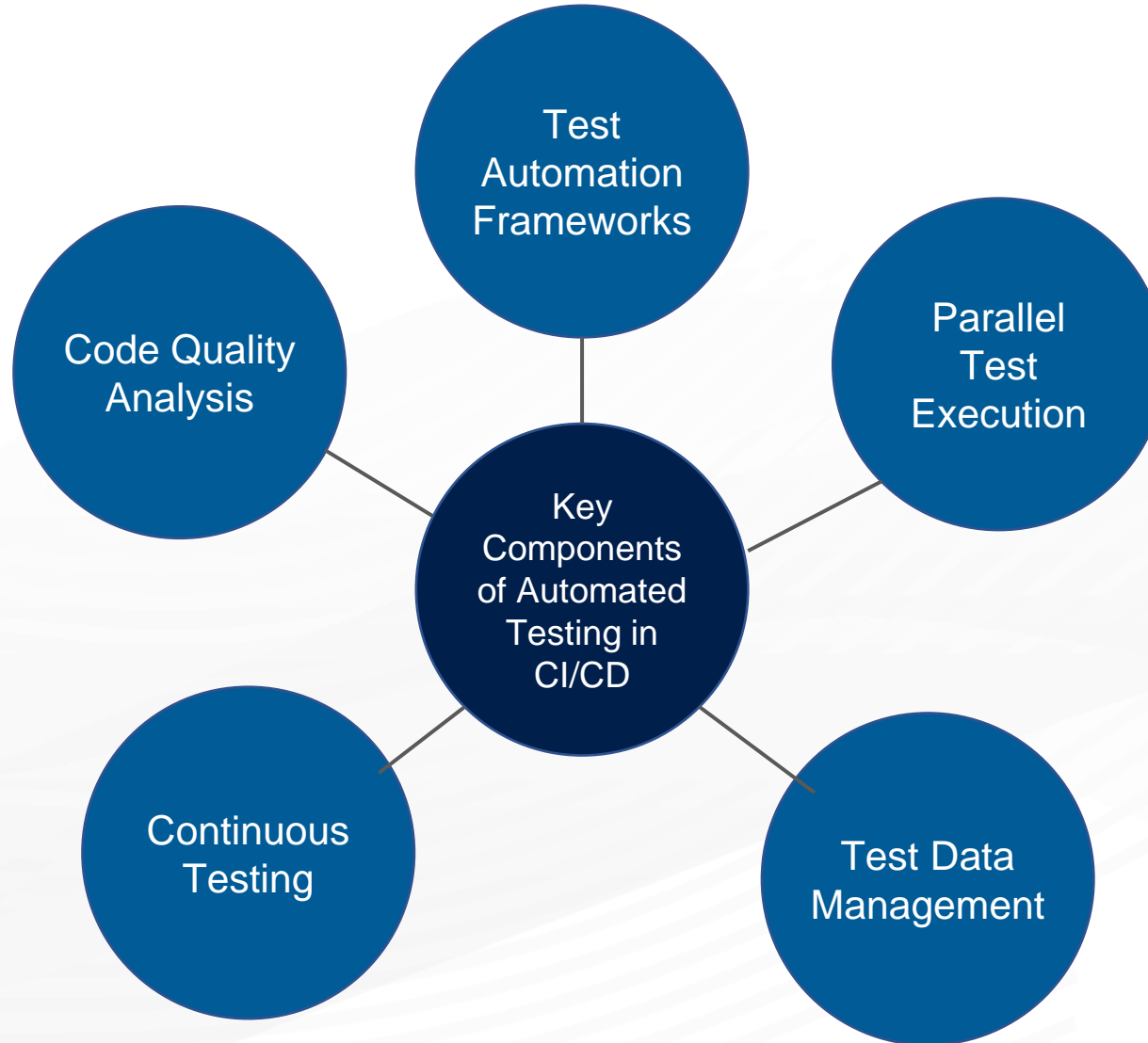
Functional Testing

Automated functional tests assess the software's functionality against specified requirements. This includes testing user interfaces, APIs, and other user-facing aspects of the application.

Performance Testing

Automated performance tests assess the application's responsiveness, stability, and scalability under varying conditions. This helps identify potential performance bottlenecks early in the development process.

3.3.8 Key Components of Automated Testing in CI/CD



3.3.9 Security Considerations in Configuration Management.

Configuration Management (CM) is a critical component of IT infrastructure that involves the systematic management of an organization's configuration items (CIs) to ensure the stability, reliability, and security of its systems.

Below are the best practices followed

- **Access Control**
- **Encryption**
- **Version Control**
- **Configuration Baselines**
- **Automation and Orchestration**

3.3.10 Access Control

Summary

- Access control is a crucial aspect of secure configuration management, limiting who can make changes to the configuration settings.
- RBAC should be used to assign permissions based on job responsibilities, ensuring that individuals have the minimum level of access required for their tasks.



3.3.10 Access Control

Best Practices

- Strict access controls to limit changes to configuration setting
- Utilize role-based access control (RBAC) to assign permissions.
- Only authorized personnel should have the necessary permissions to modify configurations.



3.3.11 Encryption

Encryption

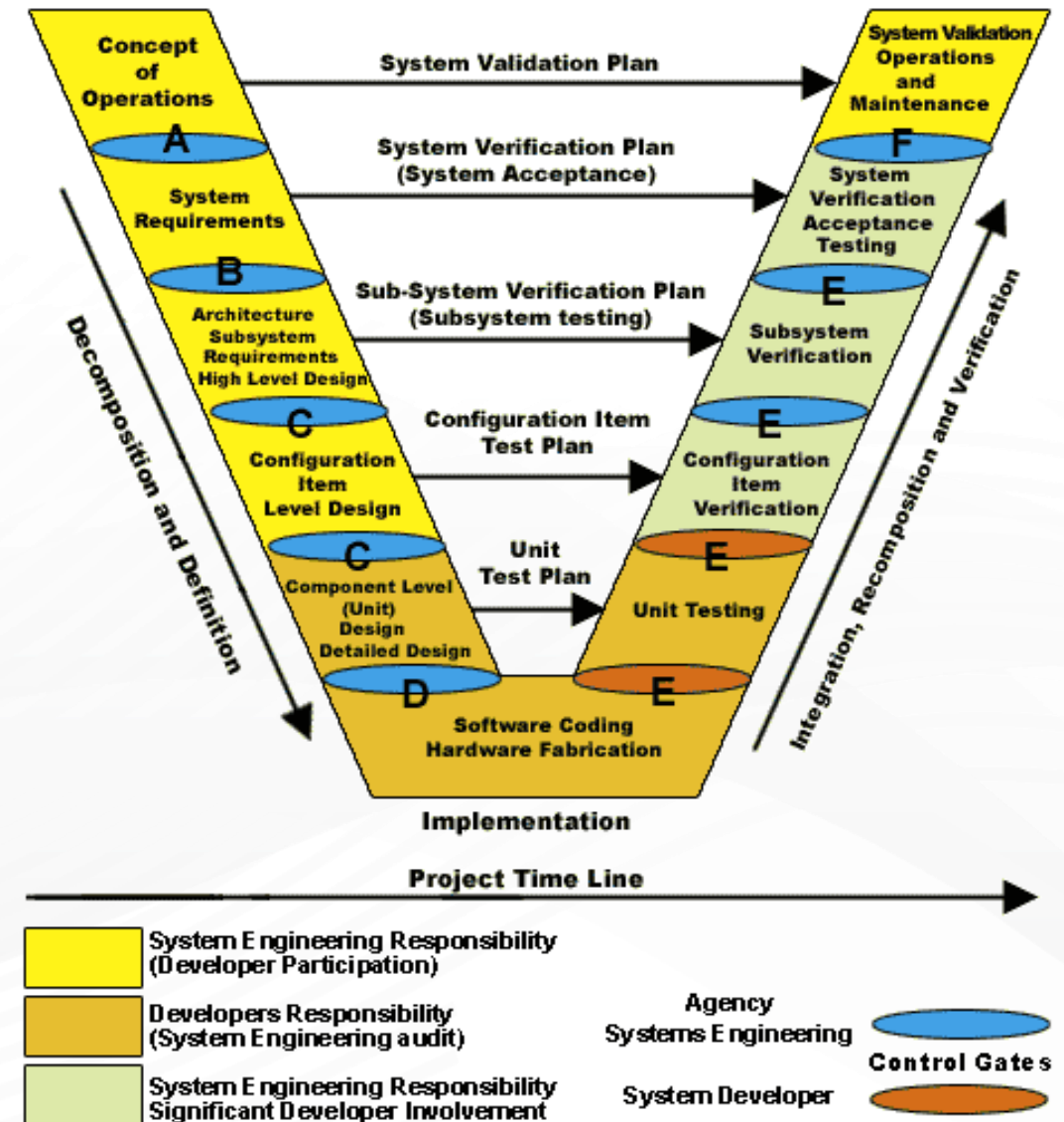
→ Encrypt sensitive configuration data to protect it from unauthorized access.



3.3.12 Configuration Baselines.

Importance of Baselines

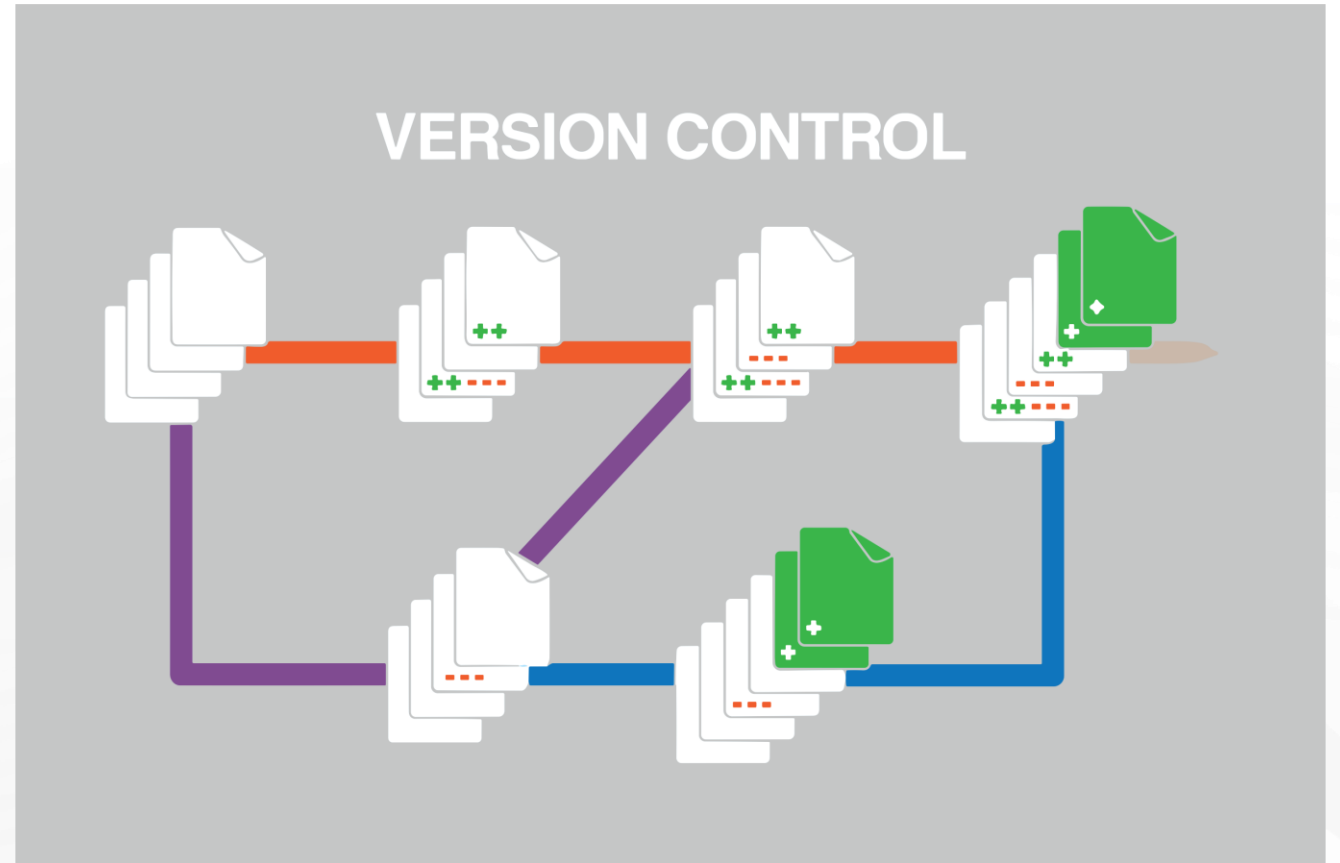
- Reference for secure configuration standards.
- Stability, reliability, and security.
- Provide a baseline for auditing
- Serve as a starting point



3.3.11 Version Control

Version Control

- Implement version control for configuration files to track changes over time. This facilitates easy rollback to previous configurations in case of errors or security incidents.
- Regularly audit and review version control logs to detect unauthorized changes and ensure accountability.



3.3.13 Automation

Benefits of Automation

- Automation ensures consistent and secure configurations across the infrastructure
- It reduces the likelihood of human errors in configuration management
- Automated processes enable rapid response to security incidents

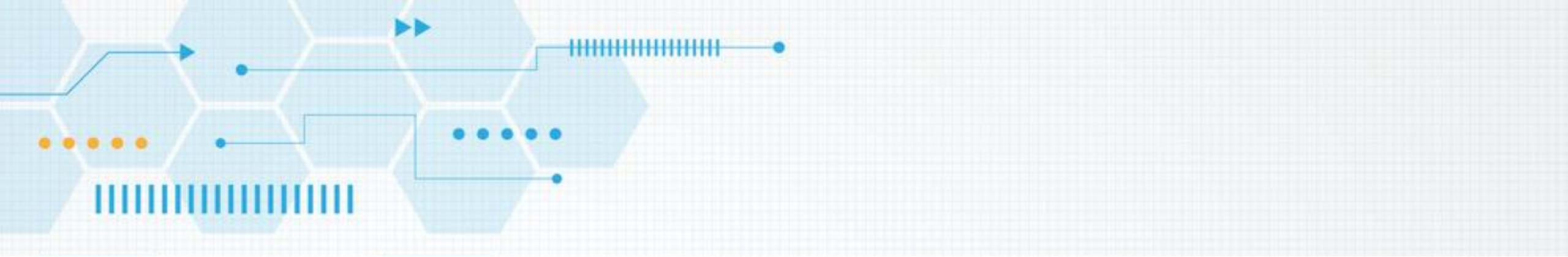


3.3.13 Orchestration

The Role of Orchestration

- Orchestration coordinates configuration changes across multiple systems
- It helps maintain synchronization and consistency in configurations
- Orchestration streamlines the process of implementing complex changes





Case Studies and Practical Implementations



3.4.1 Case Study 1

Large-scale Infrastructure Deployment

Introduction:

- This case study focuses on a large-scale infrastructure deployment undertaken by a multinational corporation (referred to as the client) seeking to revamp its existing technological backbone to meet the growing demands of a rapidly expanding user base and evolving business requirements.

[Read More](#)



3.4.2 Case Study 2

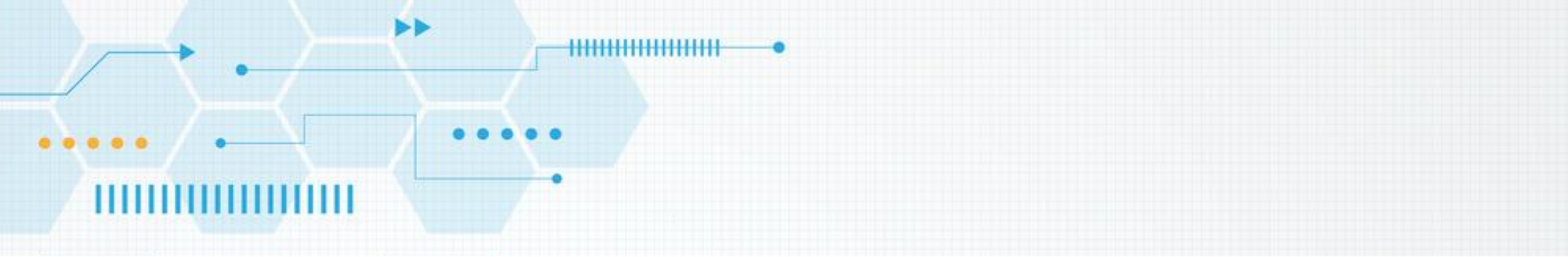
Cloud Environment Optimization

Introduction:

→ This case study provides insights into a live project undertaken by a mid-sized technology company to optimize its cloud environment. The organization aimed to streamline costs, enhance resource efficiency, and improve overall performance within its cloud infrastructure.

[Read More](#)





Hands-on Labs and Practical Exercises.



3.5.0 Hands-on Lab: Configuring Systems Using Desired State Configuration (DSC)

Objective:

To gain practical experience in configuring systems using Desired State Configuration (DSC) in a Windows environment.

Lab Environment:

Windows Server 2016 or later and Windows PowerShell 5.1 or later

- **Activity 1: Introduction to Desired State Configuration (DSC)**
- **Activity 2: Advanced DSC Configuration**
- **Activity 3: DSC with Pull Server**

[Click here to start the lab activity](#)

Refer lab 5

3.5.1 Hands-on Lab: Implementing Configuration Management with Ansible.

Objective:

To gain practical experience in implementing configuration management using Ansible.

Lab Environment:

Linux-based virtual machines or servers and Ansible installed on a control machine

- **Activity 1: Introduction to Ansible**
- **Activity 2: Advanced Ansible Playbooks**
- **Activity 3: Ansible Roles and Handlers**

[Click here to start the lab activity](#)

Refer lab 6

3.5.2 Hands-on Lab: Troubleshooting and Debugging Configuration Issues.

Objective:

To gain practical experience in troubleshooting and debugging configuration issues in a system.

Lab Environment:

Virtual machines or servers with a configuration management tool (e.g., Ansible, Puppet, Chef) installed.

Access to logs and error messages on target machines.

- **Activity 1: Identifying Configuration Issues**
- **Activity 2: Debugging with Configuration Management Tools**
- **Activity 3: Common Configuration Issues and Solutions**

[Click here to start the lab activity](#)

Refer lab 7

In a Nutshell, we learnt:



1. Principles and significance in Configuration Management for effective system maintenance.
2. Key components, including Configuration Items (CIs), Baselines, and Change Control.
3. Stability, security, and troubleshooting advantages as key benefits of Configuration Management.
4. Desired State Configuration (DSC) principles for maintaining and enforcing desired system states.
5. Popular Configuration Management Tools such as Ansible, Chef, Puppet, and SaltStack.
6. Advanced topics like drift, CI/CD integration, and security considerations for comprehensive system management.
7. Scenarios and hands-on labs for practical implementation of Configuration Management.
8. Emerging technologies and industry best practices to stay informed about future trends in Configuration Management.

End of Module

Next Module 4:

