MODULE 11 · STUDY NOTES

Creating Automated and Repeatable Deployments

Configuration management, AMI and launch-template strategy, infrastructure as code, JSON and YAML, AWS CloudFormation templates and stacks, troubleshooting, and CI/CD on AWS

Doing operations by hand does not scale: the manual configuration of distributed systems is slow, error-prone, and drifts toward inconsistency. This module is about making deployments **automated and repeatable**. It builds from the ground up — configuration management and the AWS tools that support it (user data, custom AMIs, the Chef/Puppet/Ansible frameworks, AWS OpsWorks, and AWS CloudFormation); an AMI-baking strategy and EC2 launch templates for consistent instances; the philosophy of **infrastructure as code (IaC)**, which treats an infrastructure definition as versioned source code across the whole resource lifecycle; the **JSON and YAML** formats those templates are written in; and then **AWS CloudFormation** in depth — its templates, stacks, drift detection, template sections and intrinsic functions, deployment and rollback behavior, best practices, and troubleshooting. It closes with **continuous integration and continuous delivery (CI/CD)** — the practices and AWS services (CodePipeline, CodeBuild) that carry a code change from commit to production automatically.

LEARNING OBJECTIVES

- Identify some of the AWS services for configuration management
- Describe the challenges associated with cloud deployments and potential solutions to remedy them
- Describe infrastructure as code (IaC) and the value it creates
- Describe the purpose of AWS CloudFormation
- Describe some of the types of errors with AWS CloudFormation and how to remedy them
- Describe best practices for using AWS CloudFormation
- Describe the process of troubleshooting AWS CloudFormation
- Leverage Continuous Integration / Continuous Delivery (CI/CD) deployment on AWS

11.1 Configuration management in the cloud

AWS offers several methods to configure and manage infrastructure on the cloud, and the guiding principle is to plan configuration and orchestration **proactively rather than reactively**. Deployed resources span many services — EC2 instances, Auto Scaling groups, security groups, Elastic Load Balancing load balancers, and more. Because the **manual configuration of distributed systems is time-consuming and error-prone** and leads to inconsistently configured systems, the ability to automate, monitor, and track configuration changes is the key component of any configuration management solution. Its central benefit is launching new resources quickly and consistently in an automated or semi-automated way, and it is commonly used to deploy configuration changes to running instances with increased consistency and limited downtime.

BENEFITS OF CONFIGURATION MANAGEMENT

- **Increase efficiency** — launch and change resources with less manual effort.
- **Documentation** — provides a record for every change.
- **Validation** — validate every change before release.
- **Cost reduction** — get rid of unwanted resources.
- **Security** — enforce security at every layer.
- **Deploy to running instances** — push configuration changes to instances already in service.
- **Automated and repeatable** — make the whole configuration process consistent.

AWS management tools commonly used for configuration management include **AWS CloudFormation** and **AWS OpsWorks**; **Amazon CloudWatch** complements them by monitoring resources and triggering actions when changes occur. Beyond those services, AWS provides several technologies for configuring EC2 instances that can be combined into a comprehensive strategy.

TABLE 11.1 Technologies for configuring EC2 instances

TECHNOLOGY	WHAT IT DOES
User data	Author scripts that run on instance launch — the simplest approach, but hard to manage and version at enterprise scale
Amazon Machine Images (AMIs)	Bake installations and configurations into a custom base image so instances launched from it start pre-configured, reducing deployment time
Configuration and deployment frameworks	Chef, Puppet, and Ansible — configure new instances from reusable templates and update configurations dynamically
AWS OpsWorks	A configuration management service that provides managed instances of Chef and Puppet
AWS CloudFormation	An AWS service that configures architectures for repeatable deployments

None of these techniques are mutually exclusive — they can all be used in tandem, and the right mix depends on an organization's production workflow and standards. A practical way to build a configuration strategy is to first decide what, if anything, belongs in **custom AMIs** (best reserved for foundational, slow-changing components that are standard across the organization), then choose the **configuration software** (such as CloudFormation or OpsWorks) to manage the configurable, rapidly changing components, keep everything in a **source-control and change-management system** such as Git, and then explore and experiment — developing an effective solution is an iterative process.

KEY TAKEAWAYS

- Configuration management provides multiple benefits — validation, automation, documentation, cost reduction, and security.
- AMIs can be pre-created to include required applications and files.
- AWS CloudFormation helps create, update, and delete AWS resources.
- AWS recommends managing all stack resources through AWS CloudFormation.

11.2 Using configuration software

A configuration server (such as Chef, Puppet, or Ansible) can greatly simplify common administrative tasks. The classic example is **provisioning and de-provisioning access** to EC2 instances. Suppose Jane, an engineer, holds a private key that logs her into many instances across the fleet. When Jane leaves the company, de-provisioning her access by hand is a nightmare — but with a configuration server, an administrator changes Jane's status once in the system and applies that change to every instance in the fleet with a few commands.

> **WHY A CONFIGURATION SERVER HELPS**
>
> - **Simplifies administration** — provisioning and de-provisioning access to EC2 instances across a fleet.
> - **Idempotent configuration** — resources are created or configured **once and only once**; if someone makes a manual change, the configuration server **detects it and rolls it back**, protecting instance integrity.
> - **Client-driven bootstrap** — supply **user data** to the instance to kick off client configuration; you must install the client (for example, **Knife** for Chef) plus any required configuration and security credentials, and specify which templates to run.

In a standalone deployment, the configuration server is usually its own EC2 instance that hosts a set of **templates** describing all the applications, files, and configurations needed to launch AWS resources. A web-server template, for example, might automate launching an EC2 instance, install HTTP, configure `httpd.conf`, install language environments such as PHP or Ruby, and lay down the web content directory structure; the server would hold many such recipes (MySQL server, NAT server, Windows IIS server, Maven repository, and so on). The administrator creates the templates, publishes them to the configuration server, and **checks them into source control** for version management and change tracking — and can combine custom scripting, AWS CloudFormation, AWS OpsWorks, or similar tools to describe and create the full deployment (EC2 instances, VPC configurations, databases, and other resources).

11.3 An AMI building strategy

When an organization requires that every EC2 instance start with a base set of software — in-house tools, AWS utilities, monitoring and intrusion-detection software — a good option is one or more **custom AMIs used as a base configuration**. You launch an instance from a standard AMI, pre-install and configure everything the organization requires, and then create a **custom AMI** from that instance; the new AMI becomes the source for all new instances. You can enforce the policy by scanning running instances and terminating any that were not launched from the standard AMI.

The alternative is to **configure instances at boot time**, for example by running a script through user data at launch. Most organizations settle on a **hybrid**: bake slow-changing prerequisites into a custom base AMI and configure the rest dynamically at launch. Choosing where to sit on this spectrum is a trade-off between **simplicity and flexibility**, weighed against build time, boot time, and shelf life.

TABLE 11.2 AMI baking spectrum – trade-offs

APPROACH	CONTENTS	BUILD VS. BOOT	SHELF LIFE
Full (fully baked) AMI	Application and all dependencies pre-installed	Longer build time, shorter boot time	Shorter — plan a rollback strategy
Partially configured AMI	Only prerequisite software and utilities pre-installed	Balances boot speed and build time; easier rollbacks	Longer
OS-only (JeOS) AMI	Operating system only — fully configurable at boot	Shorter build time, slower boot	Longest, most upgradeable

CREATING AN AMI – THE ESSENTIALS

- The resulting AMI **exists only in the Region** where it was created.
- By default the instance is **rebooted** during creation to ensure a consistent file system; you can override this, but AWS then cannot guarantee integrity.
- For an EBS-backed AMI, all attached volumes are captured; creation makes a **snapshot per volume**, and you are charged for the snapshots until you deregister the AMI and delete them.
- Create with the console, CLI, or API — `aws ec2 create-image` requires the **instance ID** and a **name**; Linux AMIs can also be built directly from a root-volume snapshot with `aws ec2 register-image`.

Because an AMI is anchored at the Region level, launching from it in another Region requires that you **explicitly copy** it first — with **Copy AMI** in the console or `aws ec2 copy-image` from the command line, specifying the source image ID, source Region, target Region, and a name. You can copy EBS-backed and instance store-backed AMIs, as well as encrypted AMIs (and re-encrypt with a different KMS key during the copy). A copy may fail if AWS cannot find a corresponding **Amazon Kernel Image (AKI)** in the target Region.

CREATING WINDOWS AMIS – EC2LAUNCH V2 AND SYSPREP

Before capturing a Windows AMI, run **EC2Launch v2** (included on supported Windows Server 2008-or-later instances, or downloadable). It runs once on boot, executes its configured tasks, and invokes Microsoft **Sysprep**, which strips instance-specific information so the image can be generalized. Sysprep runs three phases: **Generalize** (removes image-specific data — the security identifier, computer name, event logs, specific drivers — after which the OS is ready to become an AMI), **Specialize** (Plug and Play installs drivers and the OS generates a new computer name and SID; optional commands can run here), and **Out-of-Box Experience (OOBE)** (an abbreviated Windows Setup; an answer file automates it). You can invoke EC2Launch v2 manually after boot by restarting the service.

11.4 Amazon EC2 launch templates

A **launch template** captures the configuration needed to launch an EC2 instance so you do not have to specify it every time. Stored launch parameters include the **AMI ID, instance type, subnet, key pair, and security group**; the person who creates the template decides which options to include. Specifying a launch template at launch **reduces the risk of deployment errors** and streamlines launches for EC2 Auto Scaling, Spot Fleet, Spot Instances, and On-Demand Instances — capturing all parameters in one resource makes it easier to enforce standards, manage costs, and improve security.

LAUNCH TEMPLATE VERSIONS

- A launch template can have **one or more numbered versions**, each with different launch parameters.
- You can launch an instance from **any version**; if you do not specify one, the **default version** is used.
- You can set **any version as the default**; by default that is the **first** version.
- Example: version 1 sets instance type, AMI ID, subnet, and key pair; version 2 adds a security group; version 3 changes some values. If version 2 is the default, an unspecified launch uses version 2's parameters.

11.5 Infrastructure as code and the resource lifecycle

The key idea of **infrastructure as code (IaC)** is to encode the definition of infrastructure resources into configuration files and control their versions, **just like application software**. Its traditional role was only to *provision* resources, but every stage of the infrastructure resource lifecycle involves procedures that can be expressed as code — so IaC extends across the whole lifecycle, and each stage gains the same consistency and repeatability. AWS supplies a service for each stage.

TABLE 11.3 The infrastructure resource lifecycle

STAGE	WHAT HAPPENS	AWS SERVICE
Resource provisioning	Administrators provision resources for business needs — a repeatable process	AWS CloudFormation
Configuration management	Resources are upgraded, tuned, and patched with incremental changes	EC2 Systems Manager · OpsWorks for Chef
Monitoring and performance	Tools validate operational status via metrics, transactions, and logs	Amazon CloudWatch
Compliance and governance	Frameworks track how resources change over time to meet standards	AWS Config
Resource optimization	Administrators use performance data to optimize for cost and performance	AWS Trusted Advisor

The **provisioning** stage shows why a repeatable process matters: a release manager builds a byte-for-byte replica of production for disaster recovery; a professor instantiates an identical student environment each semester; a security administrator bakes required controls into a template so they are always present; a project manager publishes an environment template to a catalog so developers self-serve. Each situation shares the need to **instantiate resources consistently**, which is exactly what CloudFormation provides.

In the **configuration management** stage, one way to make a change is to return to provisioning, build fresh resources from the code baseline, and dispose of the old ones — an approach called **infrastructure immutability** that eliminates configuration drift. But in environments with high durability requirements, it can be preferable to make **incremental changes** to the current resources instead of reprovisioning them. Overall, IaC lets an organization automatically deploy consistently built environments and raise its operational maturity.

11.6 Describing infrastructure: JSON and YAML

CloudFormation templates are written in **JSON** or **YAML**, so understanding both formats is a prerequisite. **JSON (JavaScript Object Notation)** is a text-based, human-readable syntax for storing and transporting data as **key-value**

pairs, arrays, and other objects; its strength is exchanging complex data between computer systems, and it is the common format for RESTful APIs.

JSON BUILDING BLOCKS

- **Objects** use curly braces `{}` — an **unordered** set of key-value pairs; a colon separates each key from its value, commas separate the pairs, and the **last pair has no trailing comma**.
- **Arrays** use square brackets `[]` — an **ordered** list of values you can address by index (for example `object.additional_ingredients[1]`).
- Array values can mix types: **string, number, object, array, or Boolean**.
- Advantages: lightweight and API-friendly, easy for humans to read and machines to parse. Disadvantages: not tabular (harder to read), verbose, and **no native support for binary data** such as images.

YAML (YAML Ain't Markup Language) is a human-readable data-serialization language often used for configuration files. It provides capabilities similar to JSON — complex structures in a single document — and is portable across languages such as C, Java, Perl, Python, and Ruby. Compared with JSON it is **optimized for readability**: fewer characters, no curly braces, fewer quotation marks, **native support for comments**, and easier debugging (no hunting for a misplaced comma or brace). JSON keeps its own edge — near-universal support and usually easier generation and parsing.

YAML SYNTAX BASICS

- Use **white-space indentation** to denote structure — **never tabs**; two spaces is standard, and consistency is what matters.
- Add a **comment** with `#` at the start of the line.
- Create a **list** by starting each line with a hyphen (one item per line), or inline with square brackets and commas.
- Create an **associative array / key-value pair** with a colon followed by a space: `key: value`.
- **Strings are normally unquoted**, though single or double quotes may be used to escape characters.

THE SAME RESOURCE IN JSON AND YAML

A `WebServer` resource of `Type: AWS::EC2::Instance` with properties `ImageId`, `InstanceType: t2.micro`, `KeyName`, `SecurityGroupIds`, and `SubnetId` can be written either way. The **JSON** version wraps everything in curly braces, quotes every key and value, and separates lines with commas; the **YAML** version uses white-space indentation to show that `Type` and `Properties` are children of `WebServer`, drops the braces and most quotes, and ends lines without commas. CloudFormation accepts templates in **either** format.

11.7 AWS CloudFormation: templates, stacks, and drift

The cloud's power raises hard operational questions: How do you update servers already deployed to production? How do you consistently deploy an infrastructure across multiple Regions? How do you **roll back** a deployment that went wrong and reclaim the resources it created? How do you test and debug a deployment before production? How do you manage dependencies between systems and whole subsystems? A range of tools answer these needs — **custom scripts** that call the AWS CLI or API, **AWS OpsWorks** (managed Chef and Puppet), and **AWS CloudFormation**, which defines a collection of resources in a single JSON or YAML template and creates them together as one unit.

WHAT AWS CLOUDFORMATION DOES

- **Models and provisions** cloud infrastructure resources predictably and repeatedly.
- **Supports most AWS services** — EC2, EBS, Amazon SNS, Elastic Load Balancing, Auto Scaling, and many more.
- **Creates, updates, and deletes** a set of resources as a single unit called a **stack**.
- **Detects drift** on the stack and on individual resources. Its core objects are the **template**, the **stack**, and the **change set**.

Even when you manage resources through CloudFormation, users can still change them **outside** of it — for example, editing a stack's EC2 instance directly in the EC2 console. Such changes may be accidental or deliberate (a time-sensitive operational fix), but they complicate later stack updates and deletions. **Drift detection** identifies which stack resources have configuration changes made outside CloudFormation so you can take corrective action and bring resources back in sync with their template definitions.

KEY TERMS

Template
The specification of the AWS resources to be provisioned, written in JSON or YAML. A template can be instantiated many times.

Stack
A collection of AWS resources created from a template. Charges begin when its resources are created; on deletion, the order of deletion is determined by CloudFormation — you do not control what gets deleted when.

Change set
A summary of how proposed changes would affect running resources, reviewed before the changes are executed.

11.8 Inside a CloudFormation template

A template is organized into sections. Only **Resources** is required; the rest are optional and add flexibility. **Parameters** define name-value inputs supplied at create or update time. **Mappings** hold static two-level lookup tables — most commonly the latest AMI IDs, which vary by Region and change over time. **Init** (`AWS::CloudFormation::Init`) declares applications, files, and other resources to deploy during startup. **WaitCondition** coordinates resource creation by waiting for a signal. **Outputs** return string values — URLs, names — that are useful to users.

Parameters let you customize a template with different values every time you create or update a stack. Each parameter has a **logical name** that must be unique within the template and a supported **data type** — `String`, `Number`, `List`, `CommaDelimitedList`, an **AWS-specific parameter type**, or an **SSM parameter type**. AWS-specific types are especially useful: a parameter typed `AWS::EC2::KeyPair::KeyName` renders a dropdown of the account's key pairs when the stack is launched from the console. You reference a parameter with the `Ref` intrinsic function, and pick a single value out of a comma-delimited list with `Fn::Select`.

INTRINSIC FUNCTIONS YOU SHOULD KNOW

- **`Ref`** — references parameters and resources; also underpins using maps, joining strings, and calling other functions. Its return value **depends on the resource**: an S3 bucket returns its name, an SQS queue its URL, an EC2 instance its instance ID.
- **`Fn::GetAtt`** — returns other attributes of a resource (for example, a load balancer's `DNSName`).
- **`Fn::FindInMap`** — retrieves a value from the Mappings section.
- **`Fn::Select`** — returns one object from a list by index; **`Fn::Join`** concatenates strings; **`Fn::Base64`** encodes user data.
- **Pseudo parameters** (`AWS::Region`, `AWS::StackName`, `AWS::AccountId`) are predefined by CloudFormation — you do not declare them, and you use them with `Ref`.

Mappings match a key to a set of named values in a two-level map, and you read them with `Fn::FindInMap`. A common pattern is a map named `AWSRegionToAMI` keyed by Region name, whose values are the correct AMI ID per Region; combined with the pseudo parameter `AWS::Region`, a `Fn::FindInMap` lookup resolves to the right AMI wherever the stack runs. Note that you **cannot** include parameters, pseudo parameters, or intrinsic functions inside the Mappings section itself.

The **Resources** section declares each AWS asset to create, giving it a `Type` and optional `Properties`; multiple resources of the same type are separated by commas. Because CloudFormation's automatic sequencing rules can change as new resource types are added, it is a best practice to use the **`DependsOn`** attribute to force a consistent creation order — for example, an EC2 web server that `DependsOn` an RDS database so the database exists before the application needs it.

USER DATA (IN CLOUDFORMATION)	CLOUDFORMATION::INIT
greater control, greater risk	*declarative and cleaner*
▪ A script file, as with individual instances ▪ More control over command ordering and error handling ▪ Written in real languages with loops and conditionals ▪ Special characters (such as double quotes) must be escaped to stay valid JSON or YAML	▪ Declares the configuration you want instead of scripting procedural steps ▪ Can roll back automatically on failure ▪ Attaches metadata within the template; run by `cfn-init` ▪ Configsets let you reuse and combine config blocks across instances

`AWS::CloudFormation::Init` provides metadata that the **`cfn-init`** helper script reads from the `AWS::CloudFormation::Init` key; with it, `cfn-init` can **install packages, manage groups and user accounts, write files to disk, run commands, and enable/disable or start/stop services** on an EC2 instance.

As far as AWS is concerned, an instance is ready when it **passes its status checks** — but that does not mean your installs and configuration have finished. A **WaitCondition** acts like a stop sign that keeps the stack from being marked complete until your code signals success; the **WaitConditionHandle** is the textual handle your init code references. Your initialization code (user data or an Init script) calls **`cfn-signal`** to report success — letting CloudFormation move on — or failure, which by default rolls the stack back. You can require a specific number of success signals before the WaitCondition is fulfilled (**default is 1**) and set a `Timeout`.

The **Outputs** section returns string values created by the template that might matter to users. For example, joining `http://` with a load balancer's `DNSName` (via `Fn::GetAtt`) outputs the site's full URL. After the stack runs, outputs appear in the **Outputs tab** of the CloudFormation console and can also be retrieved through the CLI or API.

11.9 Deploying, updating, and protecting stacks

You launch a template as a stack through the **console, CLI, or API**. If an error occurs during launch, **all resources are rolled back by default** — CloudFormation treats the stack as a single unit, so every resource must be created (or deleted) successfully for the operation to succeed. If a resource cannot be created, CloudFormation rolls back and automatically deletes what it made; if a resource cannot be deleted, remaining resources are retained until the stack can be deleted cleanly.

> **PRESERVING RESOURCES AND CONTROLLING STACKS**
>
> - To preserve an **EBS volume**, set its `DeleteOnTermination` attribute to `false`.
> - To preserve any resource, set its `DeletionPolicy` attribute to `Retain`.
> - Some resources — such as a **non-empty S3 bucket** — may not be deleted when a stack is deleted.
> - Enable **termination protection** so an accidental delete request fails and the stack remains unchanged.
> - Use a **stack policy** to prevent critical resources from being unintentionally updated or replaced during a stack update.

You can override the default rollback behavior when you create a stack. The `--on-failure` option takes one of `DO_NOTHING`, `ROLLBACK`, or `DELETE`; you may specify **either `on-failure` or `disable-rollback`, but not both**. For example, `aws cloudformation create-stack --stack-name NewStack --template-body file://template.yml --on-failure DO_NOTHING` leaves any resources created by a failed stack untouched — handy for debugging. If an **update** rollback itself fails, the stack enters the `UPDATE_ROLLBACK_FAILED` state (common causes: a resource changed outside CloudFormation, insufficient permissions, limitation errors, or resources that never stabilized); after fixing the cause you run `aws cloudformation continue-update-rollback` to finish the rollback.

You edit templates in an editor that understands CloudFormation (or a JSON/YAML editor) to get parsing, auto-completion, and syntax checking — the AWS Toolkit for Visual Studio or Eclipse adds this support, and most text editors have JSON or YAML plugins. **AWS CloudFormation Designer** is a visual, drag-and-drop tool that keeps the underlying JSON or YAML in sync as you add, modify, or remove resources. Before applying changes to a running stack, create a **change set** to preview how the proposed changes would affect your resources — CloudFormation makes no changes until you execute it.

TABLE 11.4 AWS CloudFormation best practices

AREA	PRACTICES
Planning and organizing	Group resources by shared lifecycle and ownership to decide stack boundaries; reuse templates to replicate stacks across dev/test/prod; verify service quotas before launching; use AWS Organizations for consistent policies across accounts
Creating templates	Do not embed credentials — pass them as input parameters; use AWS-specific parameter types; use parameter constraints; use `AWS::CloudFormation::Init` to deploy software; validate templates before use
Managing stacks	Manage all stack resources through CloudFormation (do not change them out of band); create change sets before updating; use stack policies to protect critical resources; log calls with AWS CloudTrail; use code reviews and revision control for templates

11.10 Troubleshooting AWS CloudFormation

When CloudFormation fails to create, update, or delete a stack, you learn more from **error messages and logs**. The **AWS CloudFormation Troubleshooting Guide** is the starting point, moving from general to specific. In the console you can watch **stack events** as a stack is created, updated, or deleted; find the event that failed and expand its details, where the **status reason** often carries a message from CloudFormation or the underlying service. For EC2 problems, connect to the instance (if you can) and read the cloud-init and cfn logs.

TABLE 11.5 Common errors and how to remedy them

SYMPTOM	WHAT TO CHECK / DO
"TemplateURL must reference a valid Amazon S3 bucket."	Confirm the bucket **exists** and that you have **permission** to access it
A resource fails to create	Check IAM **permissions** for that resource type; look for a **missing required parameter**; compare against a known-good template and the documented resource syntax
A WaitCondition times out or errors	There is an error in your `CloudFormation::Init` code — analyze **`cfn-init.log`** and **`cfn-wire.log`**, collect logs via **CloudWatch Logs**, and rerun with `--on-failure DO_NOTHING` so the instance survives for inspection
Most common WaitCondition cause	Files the Init code references (scripts, MSIs) return **HTTP 403 or 404** — they cannot be accessed from the instance

WHERE THE ERRORS SURFACE

- **Template and resource-creation errors** are returned in the CloudFormation console and on stdout.
- **User data and Init errors** are written to `cloud-init.log`, `cfn-init.log`, and `cfn-wire.log`.
- Log locations: **Linux** `/var/log/` — **Windows** `C:\cfn\`.
- Retrieve logs by **logging in to the instance** or by using **Amazon CloudWatch Logs** to upload them automatically.

EXAM SCENARIO – WAITCONDITION AND CREATIONPOLICY

A Cloud Engineer is deciding when to apply the **WaitCondition** attribute in addition to **CreationPolicy**. Of the choices — automatically scaling a resource, coordinating the creation of stack resources, sending responses to resources within a VPC, tracking the configuration of EC2 instances, and tracking the status of a configuration process — the two that fit a WaitCondition are **coordinating the creation of stack resources** and **tracking the status of a configuration process**. The keywords in the scenario (coordinating creation, tracking configuration status) point straight at what a WaitCondition does.

11.11 CI/CD deployment on AWS

Continuous integration (CI) is a DevOps practice in which developers regularly **merge their code into a central repository**, after which automated builds and tests run — usually during the build/integration stage. **Continuous delivery (CD)** expands on CI: code changes are automatically built, tested, and **prepared for release**, deployed to a testing or production environment after the build stage, with a developer making the **final decision** to deploy to live production. **Continuous deployment** goes one step further and deploys to production **automatically**. The payoff is finding and fixing bugs faster, higher software quality, and shorter time to release.

TABLE 11.6 The four release-process phases

PHASE	WHAT HAPPENS
Code	Developers check changes into a source repository; peer review or pair programming provides feedback
Build	Source is compiled and quality-checked — automated unit tests, plus code metrics and style checks; container images may be created
Test	Tests that need a production-like environment — integration, load, UI, and penetration testing
Deploy	Code is deployed to production; strategies aim to reduce risk and minimize the impact of mistakes

Each phase can be **automated independently**, without automating the entire release process. The value of CI/CD stands out against the alternatives. In a **monolith**, many developers push through one shared pipeline, forcing them to coordinate changes and rebuild, retest, and redeploy the whole app — leading to painful "merge Fridays." **Non-CI/CD** processes lean on manual work: tickets and emails to request builds and deployments, which batch up and delay every change. Moving to a **service-based** model — each service with its own independent delivery pipeline — decouples teams so each can release at its own pace. This is the transformation Amazon made in 2001, breaking a monolith into services each owned end to end by a **two-pizza team** that communicates through well-defined APIs.

AWS SERVICES FOR THE SOFTWARE LIFECYCLE

- **AWS CodePipeline** — a continuous integration and delivery service that models and visualizes your release process and builds, tests, and deploys your code on **every code change**; integrates with AWS and third-party tools. Source actions poll providers such as GitHub and Amazon S3 and drop the source into a customer-owned S3 bucket.
- **AWS CodeBuild** — a fully managed CI service that compiles, tests, and packages code with **no servers to provision**.
- **Jenkins** — open-source automation software widely used for CI/CD.
- **Postman** — a convenient tool for testing a REST API.
- Benefits of CodePipeline: a configurable workflow, easy integration (Jenkins, Travis CI, Git), improved quality from a standardized process, rapid delivery, and a quick start with nothing to provision.

Module summary

- Manual configuration of distributed systems is slow and error-prone; configuration management automates, monitors, and tracks changes for consistent, repeatable deployments — supported by CloudFormation, OpsWorks, and CloudWatch.
- EC2 instances can be configured with user data, custom AMIs, frameworks (Chef/Puppet/Ansible), OpsWorks, and CloudFormation; a good strategy bakes slow-changing pieces into AMIs and configures fast-changing pieces with software, all under source control.
- Configuration servers are idempotent — they configure a resource once and roll back manual drift; a standalone server hosts templates checked into version control.
- An AMI is Region-anchored and must be copied to other Regions; choose along the fully-baked-to-OS-only spectrum by trading build time, boot time, and shelf life; Windows AMIs run EC2Launch v2 and Sysprep (Generalize, Specialize, OOBE).

- Launch templates store EC2 launch parameters (AMI ID, instance type, subnet, key pair, security group), support versions, and reduce deployment errors.
- Infrastructure as code versions infrastructure like application code and spans the whole lifecycle: provisioning (CloudFormation), configuration management (Systems Manager/OpsWorks), monitoring (CloudWatch), governance (AWS Config), and optimization (Trusted Advisor).
- CloudFormation templates are written in JSON or YAML; JSON uses braces/brackets and commas with no trailing comma, while YAML uses space indentation (never tabs), supports comments, and is less verbose.
- CloudFormation models and provisions resources from a template, instantiates them as a stack, manages them as one unit, supports most services, and detects drift when resources change outside of it.
- Only the Resources section is required; Parameters, Mappings, and Outputs are optional; intrinsic functions (Ref, Fn::FindInMap, Fn::GetAtt, Fn::Select, Fn::Join) and DependsOn wire a template together; CloudFormation::Init plus cfn-init and a WaitCondition signaled by cfn-signal coordinate instance bootstrap.
- Failed stacks roll back and delete resources by default (override with --on-failure DO_NOTHING); DeletionPolicy Retain, DeleteOnTermination false, termination protection, and stack policies protect resources; change sets preview updates; best practices cover planning, template creation, and stack management.
- Troubleshoot from console stack events and status reasons and from logs (cloud-init.log, cfn-init.log, cfn-wire.log in /var/log/ on Linux, C:\cfn\ on Windows); WaitCondition failures usually mean an Init error, often referenced files returning HTTP 403/404.
- CI/CD automates code, build, test, and deploy: continuous integration merges and tests code, continuous delivery stages releases for human approval, continuous deployment ships to production automatically — orchestrated by CodePipeline with CodeBuild, Jenkins, and Postman.

Review questions

1. List the benefits configuration management provides.

Answer: Increased efficiency; documentation for every change; validation of every change before release; cost reduction by removing unwanted resources; security enforced at every layer; the ability to deploy changes to running instances; and configuration that is automated and repeatable.

2. Why must an AMI be copied before you can launch from it in another Region, and what can make the copy fail?

Answer: An AMI is anchored to the Region where it was created, so it does not exist elsewhere until you explicitly copy it with Copy AMI or aws ec2 copy-image. The copy can fail if AWS cannot find a corresponding Amazon Kernel Image (AKI) in the target Region.

3. Which template section is required, and what are the optional sections mentioned in the module?

Answer: Resources is required. Optional sections include Parameters, Mappings, Init (AWS::CloudFormation::Init), WaitCondition, and Outputs.

4. Why is passing EC2 status checks not enough for CloudFormation, and how does your bootstrap code report real completion?

Answer: Status checks only confirm the instance is running, not that your installs and configuration finished. A WaitCondition blocks stack completion until your init code calls cfn-signal against the WaitConditionHandle to report success (or failure, which triggers a rollback).

5. What is the default behavior when a stack fails to create, and how do you override it to debug?

Answer: By default CloudFormation rolls back and deletes every resource it created. Override with --on-failure DO_NOTHING (or disable-rollback, but not both) so the resources survive and you can inspect them and their logs.

---

6. A WaitCondition times out during stack creation. Where do you look, and what is the most common cause?

Answer: It means there is an error in the CloudFormation::Init code. Analyze cfn-init.log and cfn-wire.log (collect them via CloudWatch Logs, or rerun with --on-failure DO_NOTHING and log in). The most common cause is that files the Init code references return HTTP 403 or 404.

---

7. Distinguish continuous integration, continuous delivery, and continuous deployment, and name the AWS services that support them.

Answer: Continuous integration merges code into a central repository and runs automated builds and tests. Continuous delivery also builds, tests, and stages the release but leaves the final production deploy to a human. Continuous deployment deploys to production automatically. AWS CodePipeline orchestrates the pipeline and AWS CodeBuild builds, tests, and packages code (with Jenkins and Postman also used).
