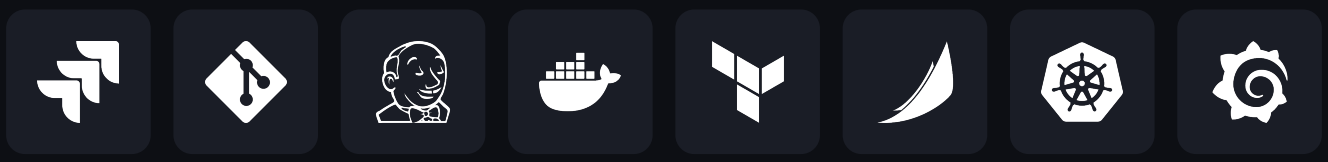
EXAM ANSWER KEY

DevOps Exam Answer Key

Question bank, Modules 1 to 6: short answers, diagrams and tool logos

From planning and version control to testing, containers, delivery and monitoring.



Contents

01 Introduction to DevOps 3

02 DevOps and Automation 13

03 Version Control Systems 22

04 Continuous Integration and Continuous Deployment (CI/CD) 28

05 Virtualization and Containerization 34

06 Continuous Testing and Configuration Management 40

Module	Topic	Questions
01	Introduction to DevOps	Q1 to Q26
02	DevOps and Automation	Q1 to Q21
03	Version Control Systems (Git)	Q1 to Q21
04	CI/CD and Jenkins	Q1 to Q21
05	Virtualization and Containerization (Docker)	Q1 to Q22
06	Continuous Testing and Configuration Management	Q1 to Q21

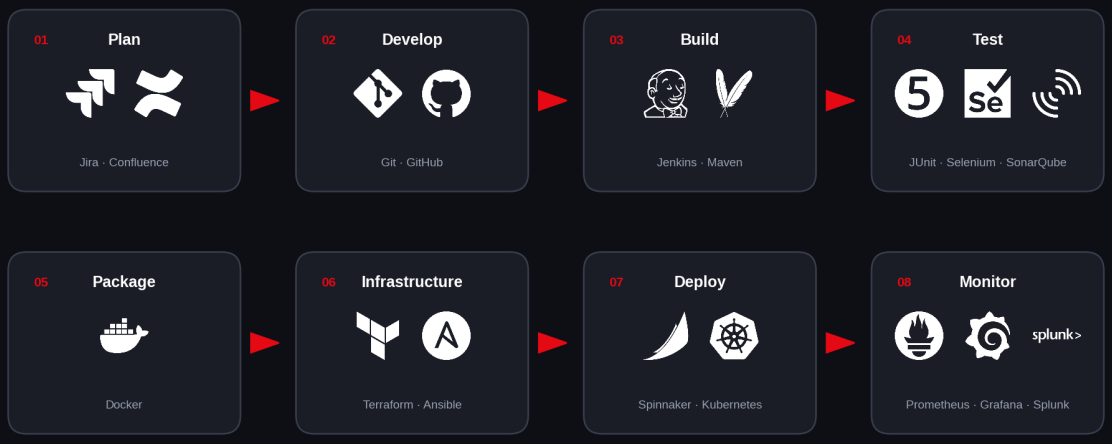
01 Introduction to DevOps

MODULE 1 • CULTURE, LIFECYCLE, PIPELINE

REMEMBER THESE TOOLS

Jira · Git · Jenkins · Docker · Terraform · Kubernetes · Prometheus · Grafana

***Memory hook: CALMS** = Culture, Automation, Lean, Measurement, Sharing. Pipeline = **Plan → Code → Build → Test → Release → Deploy → Operate → Monitor**.*



The 8-stage DevOps pipeline with the tools used at each stage

Q1 Choose an appropriate automation tool for the build phase and justify its fit.

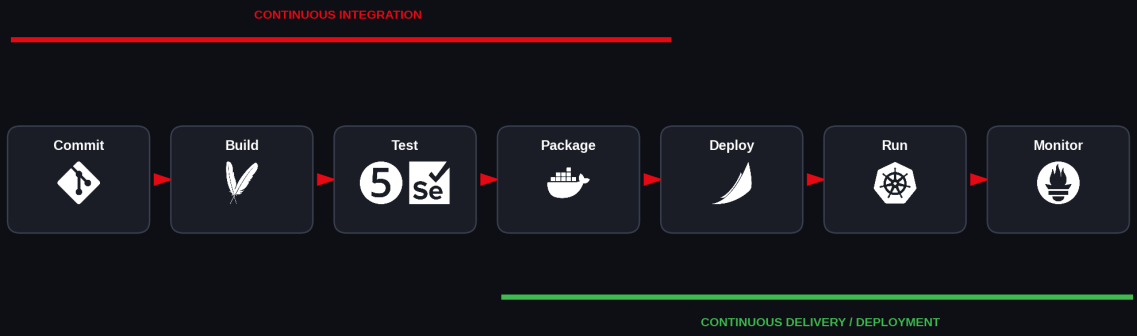
Maven (with Jenkins) for a Java project.

- **Standard lifecycle** (compile, test, package) and **automatic dependency management** from pom.xml.
- **Repeatable builds** on any machine; plugins for tests, reports and Docker.
- **Jenkins** triggers Maven on every commit. Alternatives: Gradle (faster, flexible), Ant, npm for JavaScript.

Q2 Match DevOps practices with corresponding Agile values.

Agile value	DevOps practice
Individuals and interactions over tools	Dev + Ops shared ownership, ChatOps, blameless reviews
Working software over documentation	Continuous Integration, continuous delivery, automated tests
Customer collaboration	Feedback loops, monitoring, A/B and canary releases
Responding to change	Infrastructure as code, small frequent releases, quick rollback

Q3 Demonstrate the integration flow from code commit to deployment using DevOps stages.



Commit to deployment flow

01 **Commit** code to Git (GitHub) and open a pull request.
02 **Jenkins** detects the change and starts a build with Maven.
03 **Test:** JUnit unit tests, SonarQube analysis, Selenium UI tests; the quality gate must pass.
04 **Package** into a Docker image and push it to the registry.
05 **Deploy** with Spinnaker / Kubernetes (staging, then production).
06 **Monitor** with Prometheus and Grafana; alerts feed back to the team.

Q4 Illustrate typical transitions observed in teams shifting from Agile to DevOps.

From (Agile only)	To (DevOps)
Dev and Ops in separate silos	One team, shared responsibility
Manual deployment	Automated pipeline
Release at end of sprint or quarter	Continuous delivery, many releases per day
Testing near the end	Continuous testing in the pipeline
Servers set up by hand	Infrastructure as code
Ops reacts to failures	Proactive monitoring and feedback

Q5 Examine the shift in developer responsibilities with DevOps adoption.

- **You build it, you run it:** developers help deploy and support what they write.
- Write **automated tests** and **pipeline code** (Jenkinsfile, Dockerfile).
- Define infrastructure as code and understand environments.
- **Shift-left security:** fix vulnerabilities early.
- Watch dashboards and logs, join on-call and fix production issues.

Q6 Apply DevOps principles to reduce lead time in a release cycle scenario.

Scenario: a bank app releases once a month (lead time about 4 weeks).

- **Automate** build, test and deploy in one CI/CD pipeline.
- **Small batches:** merge small changes daily (trunk-based development).
- **Parallel and automated tests**; fail fast.
- **IaC and containers** remove the wait for environments.
- **Feature flags + canary** release without waiting for big-bang approval.
- Measure **lead time for changes**; target hours instead of weeks.

Q7 Construct a deployment workflow incorporating continuous testing and integration.



Continuous testing loop inside the pipeline

01 Commit triggers CI: build and **unit tests** (JUnit).
02 Static analysis and **quality gate** (SonarQube).
03 Package Docker image; deploy to a **test environment**.
04 **Integration and UI tests** (Selenium), performance tests.
05 Approval or automatic promotion; **deploy to production**; smoke test.
06 Monitor and send results back to developers.

Q8 Justify the role of cloud-native practices within DevOps environments.

Cloud-native practice	Why it helps DevOps
Microservices	Small services deploy independently and fast
Containers + Kubernetes	Same environment everywhere, self-healing, scaling
Infrastructure as code	Environments created in minutes, repeatable
Elasticity / pay-per-use	Cost follows demand; test environments on demand
Managed services, observability	Less ops work, better feedback

Q9 Analyze the coordination between DevOps engineers and QA in a microservices project.

- **DevOps engineers** build the pipeline and environments; **QA** writes and owns automated tests that run inside it.
- **Contract tests** check that services still agree on their APIs; each service has its own pipeline and tests.
- Containers give QA **disposable test environments**; QA shifts left (tests at commit time).
- Shared dashboards: test results, failures and production monitoring feed back into new test cases.

Q10 Outline core values linked to DevOps philosophy.

Letter	Value	Meaning
C	**Culture**	Collaboration, trust, shared ownership
A	**Automation**	Build, test, deploy and configure automatically
L	**Lean**	Small batches, remove waste, flow
M	**Measurement**	Track metrics (lead time, MTTR, failure rate)
S	**Sharing**	Share knowledge, tools and feedback

Q11 List stages in a standard DevOps pipeline.

Plan, Code, Build, Test, Release, Deploy, Operate, Monitor (and then back to Plan). See the pipeline figure above.

Q12 State primary goals achieved through DevOps implementation.

- Faster, more frequent delivery and shorter lead time.
- Higher quality and fewer failures after release.
- Better collaboration between Dev and Ops.
- Reliability, quick recovery, scalability.
- Lower cost through automation; secure, repeatable processes.

Q13 Identify tools commonly used in continuous monitoring.

MONITORING TOOLS

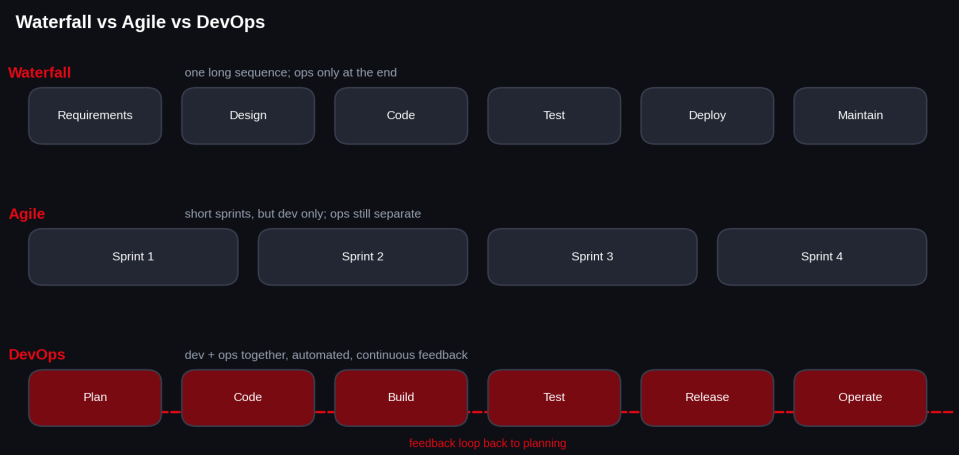
Prometheus · Grafana · Splunk · Elasticsearch · Kibana · Datadog · New Relic · Nagios

Q14 Classify deployment models aligned with DevOps in cloud environments.

Classification	Types
Cloud deployment models	Public, private, hybrid, multi-cloud
Service models	IaaS, PaaS, SaaS (also CaaS containers, FaaS serverless)
Release strategies	Blue-green, canary, rolling, recreate

Q15 Contrast DevOps lifecycle with traditional release cycles across infrastructure and testing phases.

	Traditional	DevOps
Infrastructure	Manual servers, long wait, config drift	IaC (Terraform, Ansible), on demand, identical
Testing	Manual, at the end, separate QA team	Automated, continuous, at every commit
Release	Big-bang, weeks or months	Small, frequent, automated
Feedback	Late, after go-live	Continuous from monitoring
Teams	Siloed, hand-offs	Shared ownership



Waterfall, Agile and DevOps compared

Q16 Design a secure and scalable DevOps pipeline integrating containerization and orchestration tools.



Secure (DevSecOps) pipeline with Docker and Kubernetes

- **Secure:** code scanning, dependency and image scanning, signed images, secrets vault, RBAC, audit logs.
- **Scalable:** Docker images run on Kubernetes with autoscaling, rolling updates and self-healing; IaC creates identical environments.
- **Observable:** Prometheus, Grafana and Splunk alerts close the loop.

Q17 Synthesize strategies to align DevOps with environmental sustainability in IT operations.

- **Right-size and autoscale** so servers are not idle; shut down unused test environments with IaC.
- **Containers and consolidation** raise hardware utilisation; use serverless for bursty jobs.
- **Efficient CI:** cache dependencies, run only affected tests, avoid wasted builds.
- **Green cloud:** choose low-carbon regions, schedule heavy jobs when energy is cleaner.
- **Measure** energy and carbon per deployment and set targets.

Q18 Recommend role-specific responsibilities within a multidisciplinary DevOps implementation team.

Role	Responsibility
Product owner	Priorities, backlog, acceptance
Developer	Code, unit tests, pipeline as code
QA / test engineer	Automated test suites, quality gates
DevOps engineer	CI/CD, IaC, tooling, environments
Release manager	Release planning, approvals, rollbacks
SRE / operations	Monitoring, incidents, reliability, capacity
Security engineer	Scans, policies, compliance (DevSecOps)

Q19 Evaluate performance enhancements in a system after introducing DevOps practices.

Compare the **four DORA metrics** before and after adoption (numbers below are illustrative):

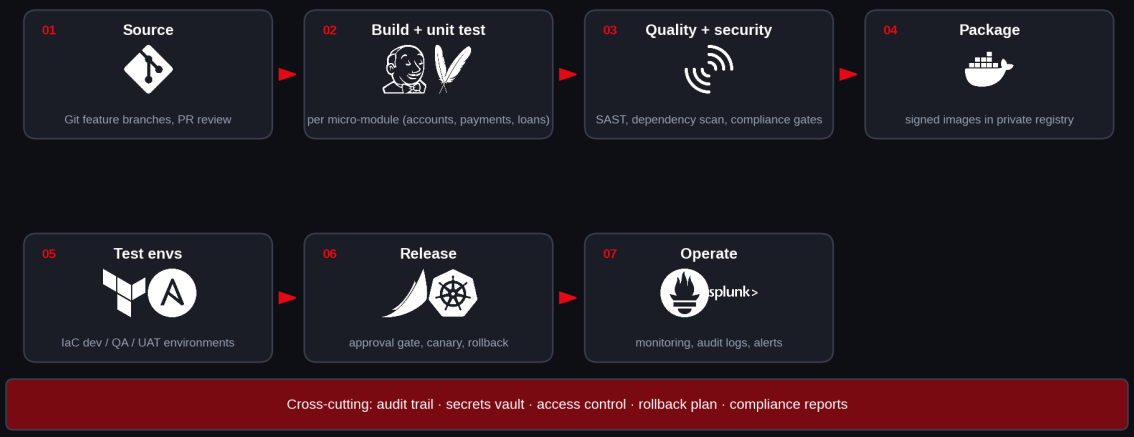
Metric	Before	After
Deployment frequency	Monthly	Daily

Metric	Before	After
Lead time for changes	4 weeks	1 day
Change failure rate	25%	8%
Mean time to recover (MTTR)	1 day	1 hour

Also compare availability, response time, defects found in production, cost per release and team satisfaction.

Q20 Create a modular DevOps framework for continuous delivery in a banking application.

Modular continuous delivery framework for a banking application



Modular continuous delivery framework for a bank

Each module (accounts, payments, loans) has its own pipeline, so one change does not block others. Compliance gates, audit trail, approvals and rollback are built in.

Q21 Formulate an ecosystem incorporating tools, practices, and policies for enterprise DevOps transformation.

Layer	Content
Policies / governance	Security and compliance policy, change management, access control, cost control
Practices	Agile, CI/CD, IaC, DevSecOps, SRE, continuous testing and monitoring
Tools	Jira, Git, Jenkins, Docker, Kubernetes, Terraform, Ansible, Prometheus (see pipeline figure)
People / culture	Training, shared ownership, communities of practice
Metrics	DORA metrics, quality, cost

Roadmap: assess the current state → pilot one team → standardise the toolchain → scale across the enterprise → continuously improve.

Q22 Define DevOps. How does it differ from traditional software development?

DevOps is a culture and set of practices that unite software **Development** and IT **Operations** to build, test, release and monitor software faster and more reliably through automation and collaboration.

	Traditional	DevOps
Teams	Separate Dev and Ops	One collaborative team
Release	Slow, big releases	Fast, small, continuous
Process	Manual	Automated
Failure handling	Blame, slow recovery	Fast feedback and recovery

Q23 What are the key objectives and benefits of adopting DevOps practices?

- **Objectives:** shorten the delivery cycle, improve quality, increase deployment frequency, reduce failures and recovery time.
- **Benefits:** faster time to market, better collaboration, higher reliability, lower cost, happier customers and teams, scalability and security.

Q24 Explain the relationship between Agile and DevOps.

Agile speeds up how teams **plan and develop** (sprints, feedback). **DevOps** extends that speed to **delivery and operations** (CI/CD, monitoring). They are complementary: Agile produces small increments, DevOps gets them to users quickly and safely. DevOps = Agile + automation + operations.

Q25 List and briefly describe the key stages of the DevOps lifecycle.

Stage	What happens	Tools
Plan	Requirements, tasks, sprints	Jira, Confluence
Code	Write and version code	Git, GitHub
Build	Compile, package	Maven, Jenkins
Test	Automated tests, quality	JUnit, Selenium, SonarQube
Release	Approve and package release	Jenkins, Docker
Deploy	Roll out to servers	Spinnaker, Kubernetes, Ansible
Operate	Run and scale	Kubernetes, Terraform
Monitor	Metrics, logs, alerts, feedback	Prometheus, Grafana, Splunk

DevOps lifecycle (infinity loop)



DevOps lifecycle as an infinity loop

Q26 What challenges can organizations face while implementing DevOps?

- **Culture:** resistance to change and silos.

- **Skills gap:** new tools and practices to learn.
- **Legacy systems** that are hard to automate or containerise.
- **Tool sprawl** and integration effort.
- **Security and compliance** concerns; initial cost; difficulty measuring success.

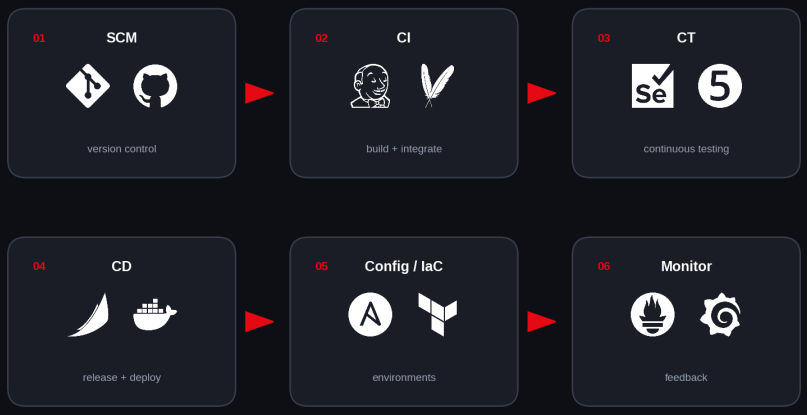
02 DevOps and Automation

MODULE 2 • TOOLCHAIN, CI, CONFIG, CONTAINERS, MONITORING

REMEMBER THESE TOOLS

Git | Jenkins | Selenium | Ansible | Terraform | Docker | Kubernetes | Prometheus

***Memory hook: SCM → CI → CT → CD → Config → Container → Monitor**. Automation removes manual, error-prone steps.*



Toolchain: version control, CI, continuous testing, delivery, config and monitoring

Q1 Define actions performed in version control branching.

- **Create** a branch (git branch / checkout -b), **switch** between branches.
- **Commit** work on the branch, **push** it to the remote.
- **Merge** back (pull request, review), **resolve conflicts**, **delete** the finished branch.

Q2 Name two benefits achieved through automated configuration scripts.

- **Consistency:** every server is configured the same way (no configuration drift).
- **Speed and fewer errors:** environments are built in minutes, repeatably, and are version controlled.

Q3 List major stages involved in continuous integration pipelines.

Commit → Checkout → Build/Compile → Unit test → Static analysis → Package/Artifact → Report/Notify.

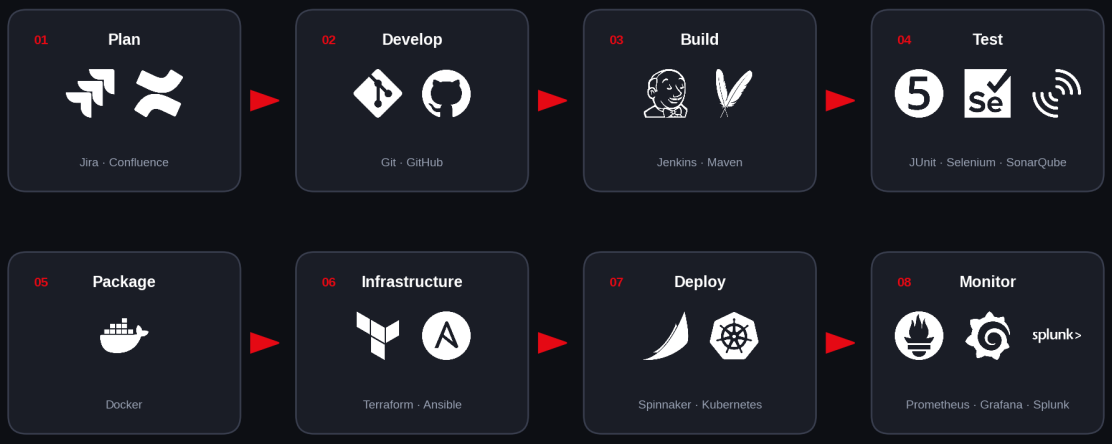
Q4 Recognize suitable file types managed by version control tools.

Best for **text files**: source code (.java, .py, .js), configuration (.yml, .json, .xml, .properties), scripts (.sh), documentation (.md), Dockerfile, Jenkinsfile, Terraform and Ansible files. Large **binaries** (videos, builds) are better stored in artifact repositories (or Git LFS).

Q5 Classify container orchestration platforms based on feature sets.

Platform	Features / type
Kubernetes	Most complete: auto-scaling, self-healing, rolling updates, service discovery; open source
Docker Swarm	Simple, built into Docker, lightweight clustering
HashiCorp Nomad	Simple scheduler for containers and non-containers
OpenShift	Kubernetes plus enterprise security and developer tools
Amazon ECS / EKS	Cloud managed orchestration

Q6 Identify matching tools for each DevOps phase in a pipeline.



Tool for each DevOps phase

Q7 Suggest an open-source tool for log aggregation during monitoring.

ELK / Elastic Stack (Elasticsearch, Logstash, Kibana); alternatives: Fluentd, Graylog, Grafana Loki.

Q8 Describe the steps in establishing a version control workflow in a team.

01 Choose a VCS (Git) and host (GitHub / GitLab).
02 Create the repository and protect the **main** branch.
03 Agree a **branching strategy** (feature branches, Git Flow or trunk-based).
04 Developers **clone**, create a branch, commit often.
05 Open a **pull request**; automated checks and code review.
06 **Merge**, tag releases, and delete the branch.

Q9 Analyze the transition from traditional deployment to continuous methods.

Traditional	Continuous
Manual copy to servers, weekend releases	Automated pipeline, any time
Large, risky releases	Small, low-risk releases
Long feedback	Immediate feedback
Rollback is hard	Quick rollback or roll-forward

Q10 Design a toolchain that connects integration, testing, and deployment.

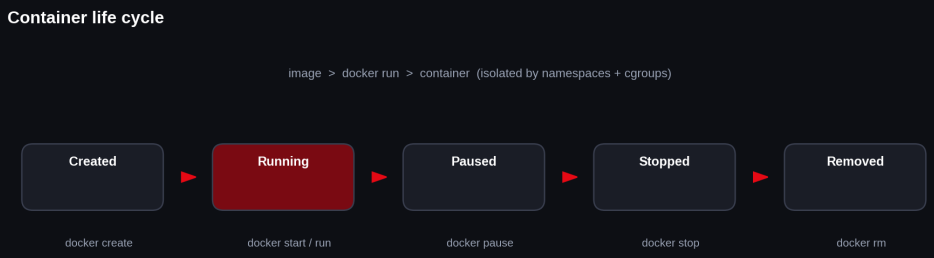
Git (commit) → **Jenkins** (build, trigger) → **Maven + JUnit** (unit tests) → **SonarQube** (quality) → **Selenium** (UI tests) → **Docker** (package) → **Spinnaker / Kubernetes** (deploy) → **Prometheus + Grafana** (monitor). Each tool triggers the next, and a failure stops the pipeline.

Q11 Evaluate configuration drift in cloud-managed environments.

- **Drift** = servers slowly differ from their intended configuration because of manual changes.
- **Problems:** "works on one server only", security gaps, failed deployments, hard debugging.
- **Detect:** compare actual vs desired state (terraform plan, Ansible check mode, cloud config rules).

- **Prevent:** infrastructure as code, immutable servers/containers, read-only access, pipeline-only changes.

Q12 Illustrate stages of container lifecycle and isolation benefits.



Container life cycle

- **Isolation** (Linux namespaces + cgroups): each container has its own processes, network and limited CPU/memory.
- **Benefits:** no dependency conflicts, security boundary, consistent behaviour, many containers per host.

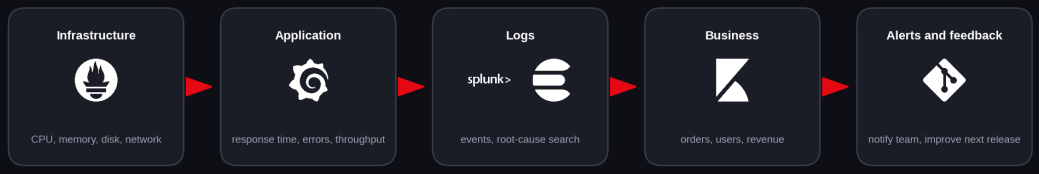
Q13 Choose tools for implementing configuration management in hybrid infrastructure.

CHOICES

Ansible Terraform Puppet Chef

Ansible (agentless, works on-prem and cloud over SSH) to configure servers, **Terraform** to provision cloud and on-prem resources, with **Puppet/Chef** for large agent-based fleets.

Q14 Map each monitoring layer with appropriate feedback metrics.



Monitoring layers

Layer	Metrics	Tool
Infrastructure	CPU, memory, disk, network	Prometheus, Nagios
Application	Response time, error rate, throughput	Grafana, New Relic
Logs	Error events, traces	Splunk, ELK
Business	Orders, users, revenue	Kibana, dashboards

Q15 Build a version control structure for a distributed DevOps team with branching logic.



Branching structure

- **main** = always releasable; **develop** = integration; **feature/*** short branches; **release/*** and **hotfix/*** as needed.
- Pull requests with reviews and CI checks; protected main; tags for releases; naming rules and small, frequent merges for distributed teams across time zones.

Q16 Critique pipeline failures in CI/CD systems and trace root causes with examples.

Failure	Root cause (example)	Fix
Build fails	Missing dependency or wrong Java version	Pin versions, use containers for builds
Tests fail randomly	Flaky UI test, shared test data	Stabilise tests, isolate data
Works locally, fails in CI	Environment difference	Docker-based builds, IaC
Deploy fails	Wrong credentials / config	Secrets manager, config as code
Slow pipeline	Long sequential tests	Parallelise, cache

Q17 Compare stateless vs stateful containerized deployments in production systems.

	Stateless	Stateful
Data	No local data; state in external DB	Keeps data (database, queue)
Scaling	Easy, add or remove copies	Harder (replication, ordering)
Recovery	Restart anywhere	Needs persistent volumes
Kubernetes	Deployment	StatefulSet + PersistentVolume
Example	Web / API service	MySQL, Kafka

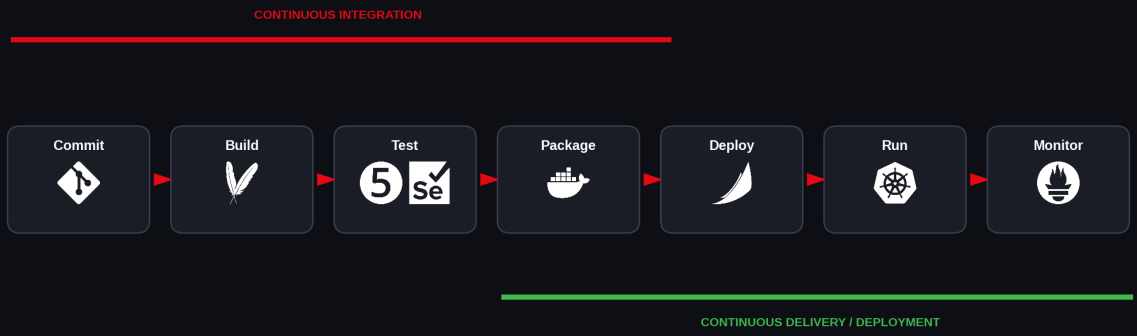
Q18 Design a continuous testing strategy using multiple testing frameworks.

Level	Framework	When
Unit	JUnit	Every commit
Static / quality	SonarQube	Every commit
API / integration	Postman, REST Assured	After build
UI	Selenium	Deployed test environment
Performance	JMeter	Nightly / pre-release
Security	OWASP ZAP, dependency scan	Pipeline stage

Q19 Evaluate resource consumption patterns from continuous monitoring dashboards.

- Read CPU, memory, disk and network over time (Grafana); find **peaks, steady growth (leaks), idle servers**.
- **Actions:** autoscale at peaks, right-size over-provisioned servers, fix memory leaks, set alerts and capacity plans, cut cost.

Q20 Compose a tool pipeline integrating SCM, CI, CT, CD, containerization, and monitoring.



SCM to monitoring pipeline

SCM: Git/GitHub → **CI:** Jenkins + Maven → **CT:** JUnit, Selenium → **Container:** Docker → **CD:** Spinnaker + Kubernetes → **Monitor:** Prometheus + Grafana.

Q21 Recommend policies to secure configuration files and prevent unauthorized modifications.

- Keep configs in **version control** with **branch protection and mandatory reviews**.
- Never store **secrets** in files; use a vault (HashiCorp Vault, cloud secrets manager).
- **Role-based access** and least privilege; sign commits; audit logs.
- Encrypt files at rest, change only through the pipeline, run **drift detection** and policy-as-code checks.

03 Version Control Systems

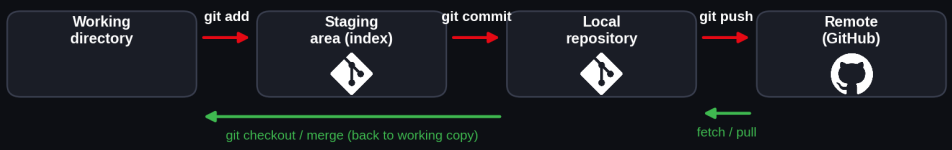
MODULE 3 • GIT, CVCS VS DVCS, WORKFLOWS

REMEMBER THESE TOOLS

Git · GitHub · GitLab · Bitbucket · Mercurial · Subversion · CVS

***Memory hook: add → commit → push** (send), **fetch → merge / pull** (receive). Git = **distributed**, CVS/SVN = **centralised**.*

Git: the four areas and the commands between them



git clone copies a remote repository; git status shows the state of the areas

Git areas and the commands that move changes between them

Q1 List two characteristics of distributed version control systems.

- Every developer has a **full copy of the repository and its history** locally.
- Can **work offline**; there is no single point of failure; branching and merging are fast.

Q2 State two industry use cases where Git enhances code management.

- **Team collaboration:** feature branches, pull requests and code reviews on GitHub / GitLab.
- **CI/CD triggers and releases:** every push starts a Jenkins build; tags mark release versions and allow rollback.

Q3 Identify two key differences between Git and CVS.

	Git	CVS
Architecture	Distributed (full local repo)	Centralised server
Commits	Atomic, whole-project snapshots	File-by-file revisions
Branching	Cheap and fast	Slow, painful
Offline work	Yes	No (needs the server)

Q4 Match Git commands with their respective functions in a repository workflow.

Command	Function
git init	Create a new repository
git clone URL	Copy a remote repository
git status	Show changed / staged files
git add .	Stage changes
git commit -m "msg"	Save staged changes to local history
git push	Upload commits to the remote
git pull	Fetch and merge remote changes
git branch / checkout -b	List / create and switch branch
git merge branch	Combine a branch into the current one
git log	Show commit history

Q5 Classify commands under local and remote operations in Git.

Local operations	Remote operations
init, add, commit, status, log, diff, branch, checkout, merge, stash, reset	clone, remote add, fetch, pull, push

Q6 Select branching models suited for collaborative development.

Git Flow (scheduled releases), **GitHub Flow** (continuous deployment), **Trunk-based development** (very frequent merges to main), **Forking workflow** (open source), **GitLab Flow** (environment branches).

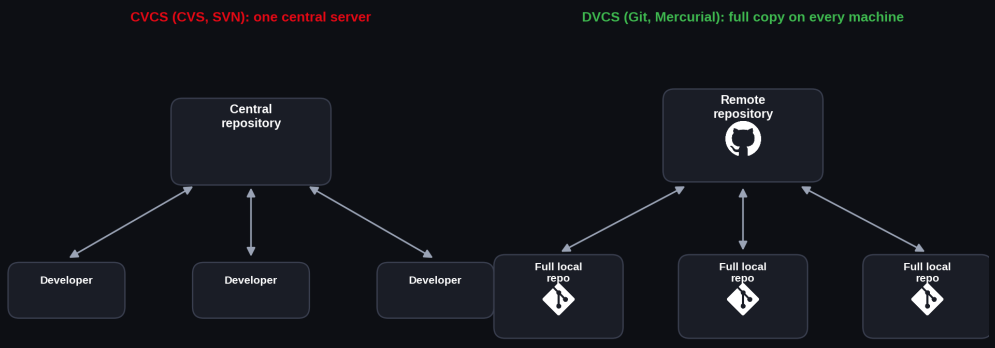
Q7 Point out one feature that distinguishes Mercurial from Git.

Mercurial has a **simpler, more consistent command set and no staging area** (you commit directly), while Git has the index/staging area and more flexible, powerful history editing.

Q8 Describe Git's role in modern software development pipelines.

- **Single source of truth** for code and infrastructure files.
- A push or pull request **triggers CI/CD** (webhook to Jenkins).
- Enables **collaboration, review, traceability and rollback**; basis of **GitOps**.

Q9 Compare CVCS and DVCS in terms of scalability and collaboration.



Centralised vs distributed version control

	CVCS	DVCS
Scalability	Server is a bottleneck	Load spread, local operations are fast
Collaboration	Needs constant server access, locks/conflicts at commit	Work independently, share by push/pull, review via pull requests
Failure	Server down means work stops	Any clone can restore the repository

Q10 Demonstrate the usage of Git commands to clone, commit, push, and pull from a repository.

```
git clone https://github.com/user/app.git   # copy the remote
cd app
echo "hello" > readme.txt
git add readme.txt                          # stage
git commit -m "Add readme"                  # commit locally
git push origin main                        # upload
git pull origin main                        # get others' changes
```

Q11 Analyze a use case involving Git branching strategy in a feature-release cycle.

Use case: an e-commerce team adds a “wishlist” feature before a release.

01 Create **feature/wishlist** from develop; commit and push.
02 Open a **pull request**; CI runs tests, reviewers comment.
03 Merge into **develop**; the team tests the integrated build.
04 Cut **release/1.4**; only bug fixes go in.
05 Merge release into **main**, tag **v1.4**, and merge back into develop.
06 Urgent bug: **hotfix/*** from main, merged to both.



Feature and release branching

Q12 Organize a table comparing Git workflows like Git Flow, GitHub Flow, and Forking Workflow.

	Git Flow	GitHub Flow	Forking Workflow
Branches	main, develop, feature, release, hotfix	main + short feature branches	Each contributor has own fork
Release style	Planned, versioned	Continuous, deploy from main	Maintainer merges pull requests
Complexity	High	Low	Medium
Best for	Scheduled releases, large teams	Web apps, CI/CD teams	Open source projects

Q13 Illustrate the sequence of Git operations involved in working with remote repositories.

01 **git clone** (or **git remote add origin URL**).
02 **git fetch** / **git pull** to update.
03 Work: edit, **git add**, **git commit**.
04 **git push origin branch** to publish.
05 Open a pull request; after merge, **git pull** again.

Q14 Distinguish Git from Mercurial based on branching, ease of use, and platform integration.

	Git	Mercurial
Branching	Lightweight pointers; very flexible	Named branches, bookmarks; more rigid
Ease of use	Powerful but steeper learning curve	Simpler, consistent commands
Platform integration	Huge: GitHub, GitLab, Bitbucket, all CI tools	Smaller ecosystem (Bitbucket dropped support)

Q15 Construct a branching and merging strategy using Git for a multi-developer project.

- **main** protected and always deployable; **develop** integration branch; one **feature/*** branch per task.
- Developers **rebase or merge develop** into their branch daily to avoid big conflicts.
- Merge only through **pull requests** with review and passing CI; **squash** or merge commit by team rule.
- Tag releases; delete merged branches.

```
git checkout -b feature/login develop
# ...work, commit...
git fetch origin && git merge origin/develop     # stay current
git push -u origin feature/login                 # then open a pull request
```

Q16 Evaluate the pros and cons of CVCS and DVCS architectures for enterprise applications.

	Pros	Cons
CVCS	Simple, central control and access, easy to understand, locking for binaries	Single point of failure, needs network, slow branching
DVCS	Offline work, speed, redundancy, powerful branching, better for distributed teams	Larger clones, more to learn, needs policies for central control

Q17 Recommend a suitable Git workflow for a fast-paced startup environment with justification.

GitHub Flow / trunk-based development. Short-lived feature branches, pull request, automated tests, merge to main and deploy immediately. **Why:** minimal overhead, fast feedback, easy continuous deployment, small changes keep risk low.

Q18 Create a comparison report between Git, Mercurial, and CVS based on key DevOps requirements.

Requirement	Git	Mercurial	CVS
Architecture	Distributed	Distributed	Centralised
Branching / merging	Excellent	Good	Poor
CI/CD integration	Best (webhooks, all tools)	Limited	Very limited
Collaboration (PR)	GitHub / GitLab	Limited	None
Atomic commits	Yes	Yes	No
Community / support	Largest	Small	Obsolete

Q19 Create a quick reference guide for beginners to effectively use Git repositories and manage branches.

Task	Command
Start	git init | git clone URL
Check	git status | git log --oneline | git diff
Save	git add file | git commit -m "msg"
Branch	git branch | git checkout -b name | git switch name
Merge	git merge name | git branch -d name
Share	git push -u origin name | git pull
Undo	git restore file | git reset HEAD~1 | git revert id
Shelve	git stash | git stash pop
Release	git tag v1.0 | git push --tags

Q20 Interpret error messages from Git CLI and propose resolution strategies for each.

Error	Meaning	Fix
fatal: not a git repository	Not inside a repo	cd to the repo or git init
CONFLICT (content): Merge conflict	Same lines changed in two branches	Edit the file, remove markers, git add, git commit
rejected ... non-fast-forward	Remote has new commits	git pull --rebase, then push
Permission denied (publickey)	SSH key not set up	Add SSH key to GitHub or use HTTPS
You are in 'detached HEAD' state	Checked out a commit, not a branch	git switch -c new-branch
nothing to commit	No staged changes	git add the files first
pathspec did not match	Wrong file or branch name	Check spelling with git status / git branch

Q21 Formulate a migration plan to switch from CVS to Git in a legacy enterprise system.

01 **Assess:** inventory repositories, size, branches, users, and build scripts.
02 **Plan:** choose Git host and workflow; train the team; freeze window.
03 **Convert history** with tools such as **cvs2git** or git cvsimport; map CVS authors to Git users.
04 **Verify:** compare tags, branches and file contents.
05 **Update integrations:** CI/CD, issue tracker, access control.
06 **Cut over:** make CVS read-only, run both briefly, then decommission CVS.

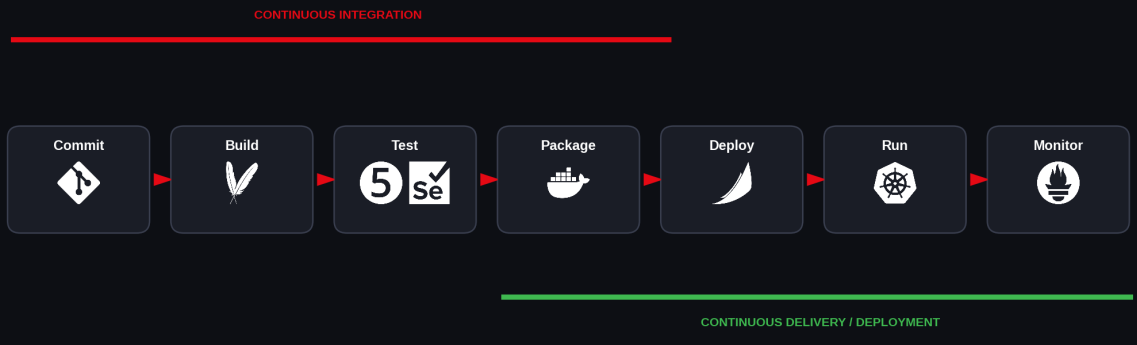
04 Continuous Integration and Continuous Deployment (CI/CD)

MODULE 4 • JENKINS, PIPELINE, JENKINSFILE

REMEMBER THESE TOOLS

Jenkins | GitHub | Maven | JUnit | SonarQube | GitLab CI | CircleCI | GitHub Actions

> ***Memory hook: CI** = merge often + build + test automatically. **Delivery** = ready to release (manual approval). **Deployment** = released automatically.*



CI/CD pipeline

Q1 List core activities performed during a Jenkins build job.

Checkout source from Git → **build/compile** (Maven) → **run tests** (JUnit) → **analyse** code → **archive artifacts** → **publish reports** → **notify** (email/Slack) → optional deploy.

Q2 State key benefits achieved through automation in CI/CD pipelines.

- Faster, more frequent releases; **early bug detection**.
- Consistent, repeatable builds with **fewer manual errors**.
- Less integration pain, higher quality, quick feedback, saved effort.

Q3 Identify two CI tools other than Jenkins used in industry environments.

OTHER CI TOOLS

GitLab CI | CircleCI | GitHub Actions | Travis CI

Q4 Match CI/CD stages with corresponding DevOps goals.

Stage	DevOps goal
Commit / source	Collaboration, traceability
Build	Consistency, automation
Test	Quality, early feedback
Package / release	Repeatable artifacts
Deploy	Speed, low risk
Monitor	Reliability, continuous feedback

Q5 Select configuration options used in Jenkins job scheduling.

- **Build periodically** (cron syntax, e.g. **H 2 * * *** = nightly).
- **Poll SCM** (e.g. **H/5 * * * *** checks Git every 5 minutes).
- **GitHub hook trigger** (webhook on push) and **Build after other projects**.
- **Trigger builds remotely** via URL token; manual "Build Now".

Q6 Classify build failure types observed in Jenkins console output.

Type	Example
Compilation	Syntax error, missing class
Test failure	JUnit assertion failed
Dependency	Cannot download a library
Configuration / environment	Wrong JDK, missing credentials
Infrastructure / timeout	Agent offline, out of disk
Quality gate	SonarQube threshold not met

Q7 Choose steps involved in integrating Git repositories with Jenkins.

01 Install the **Git** and **GitHub** plugins; add credentials.
02 Create a job; set **Source Code Management = Git** with the repository URL and branch.
03 Enable a trigger: **GitHub webhook** or Poll SCM.
04 Add build steps and post-build actions; save and run.

Q8 Explain stages of a Jenkins-managed CI/CD pipeline using a real-world use case.

Use case: an online shopping Java web app.

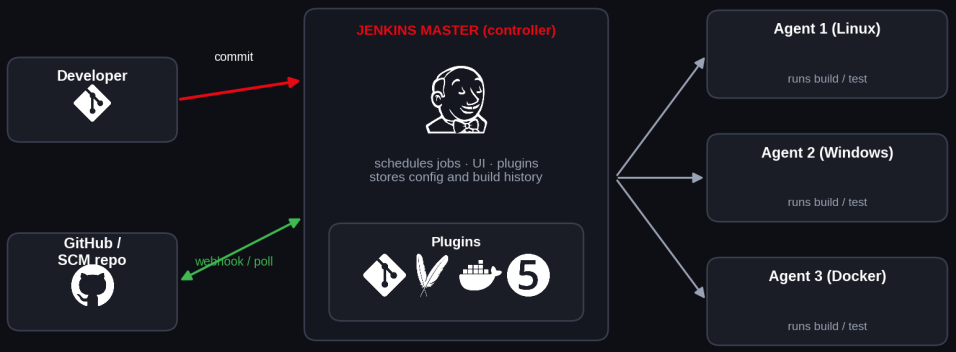
01 **Checkout** from GitHub on every push.
02 **Build** with Maven.
03 **Test:** JUnit, SonarQube analysis.
04 **Package** a Docker image and push to registry.
05 **Deploy** to staging; run Selenium tests.
06 **Approve and deploy** to production; notify the team.

Q9 Analyze the impact of continuous integration on software release frequency.

CI merges small changes often and tests each automatically, so integration problems are found within minutes, the main branch stays stable and the product is always **ready to release**. Teams move from rare big releases to **frequent small ones** (weekly or daily), with lower risk and faster feedback.

Q10 Illustrate Jenkins architecture including master-agent model and plugin system.

Jenkins architecture: master and agents



Jenkins master-agent architecture

- **Master (controller):** schedules jobs, manages configuration, UI and build history.
- **Agents (nodes):** run the builds, can be Linux, Windows or Docker; allow parallel builds.
- **Plugins:** add features (Git, Maven, Docker, JUnit, Pipeline, Blue Ocean).

Q11 Compare scheduling strategies for running CI jobs during development and release cycles.

	During development	During release
Trigger	On every commit (webhook), plus nightly full build	Scheduled release builds, manual or tag-based
Scope	Fast unit tests	Full regression, performance, security
Goal	Quick feedback	Confidence and stability

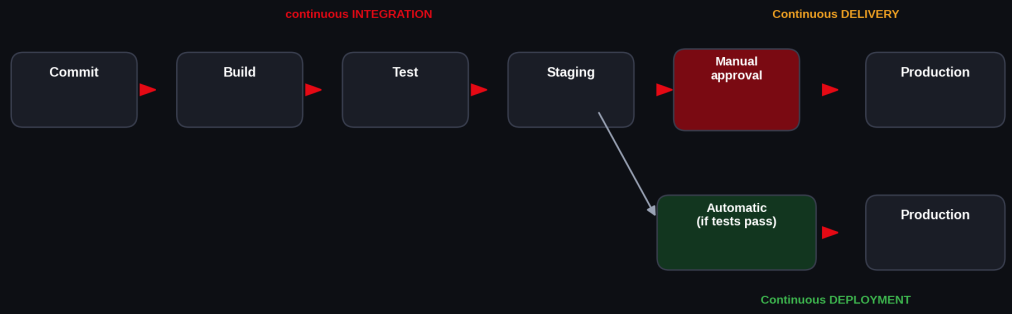
Q12 Construct a Jenkins job configuration involving Git-based trigger and post-build steps.

01 New Item → **Freestyle project** (or Pipeline).
02 **SCM:** Git, URL, branch */main, credentials.
03 **Build triggers:** GitHub hook trigger for GITScm polling.
04 **Build step:** Invoke top-level Maven targets: **clean package**.
05 **Post-build actions:** publish JUnit test report, archive artifacts (*.jar), e-mail notification, trigger the deploy job.

Q13 Evaluate the reliability of Jenkins builds through reporting plugins and test logs.

- **JUnit plugin:** pass/fail counts and trend graph; **SonarQube plugin:** quality gate.
- **Console output and build history** show failures and flaky builds; **Blue Ocean** visualises stages.
- Reliable = stable pass rate, short build time, reproducible results, few flaky tests.

Q14 Distinguish continuous deployment from continuous delivery based on risk management.



Continuous delivery has a manual gate; continuous deployment is fully automatic

	Continuous delivery	Continuous deployment
Production release	**Manual approval**	**Automatic** if tests pass
Risk control	Human check before release	Relies on strong tests, canary and monitoring
Risk level	Lower per release	Higher unless tests are excellent

Q15 Build a CI/CD pipeline using Jenkins for a multi-module application including GitHub integration.

```
pipeline {
  agent any
  stages {
    stage('Checkout') { steps { git url: 'https://github.com/org/shop.git', branch: 'main' } }
    stage('Build')    { steps { sh 'mvn -B clean package' } }   // builds all modules
    stage('Test')     { steps { sh 'mvn test' } post { always { junit '**/target/surefire-reports/*.xml' } } }
    stage('Docker')   { steps { sh 'docker build -t shop:${BUILD_NUMBER} .' } }
    stage('Deploy')   { steps { sh 'kubectl apply -f k8s/' } }
  }
  post { failure { mail to: 'team@x.com', subject: 'Build failed', body: '${BUILD_URL}' } }
}
```

A GitHub webhook triggers the pipeline on each push; Maven builds all modules from the parent pom.

Q16 Evaluate Jenkins capabilities in reducing integration errors during parallel development.

- Builds and tests **every commit and pull request**, so conflicts and breaking changes appear within minutes.
- **Parallel builds** on agents and a **quality gate** keep the main branch green.
- Notifications make the owner fix the break immediately.

Q17 Compose a plan to automate test and deployment jobs using Jenkins pipeline DSL and cron triggers.

```
pipeline {
  agent any
  triggers { cron('H 2 * * *') ; pollSCM('H/5 * * * *') }   // nightly + every 5 min
  stages {
    stage('Test')   { steps { sh 'mvn test' } }
    stage('Deploy') { steps { sh './deploy.sh staging' } }
  }
}
```

Q18 Compare Jenkins with other CI tools in terms of plugin ecosystem, scalability, and security.

	Jenkins	GitLab CI	GitHub Actions	CircleCI
Plugins	1800+ plugins, biggest	Built in features	Marketplace actions	Orbs
Scalability	Master + many agents	Runners	Hosted / self-hosted runners	Cloud, scales easily
Security	Self-managed; needs hardening	Integrated with GitLab	Integrated, secrets store	Managed, SOC compliance
Setup	Self-hosted, more work	Easy with GitLab	Easiest with GitHub	Easy (SaaS)

Q19 Formulate a troubleshooting guide for failed builds in a CI/CD pipeline involving Git and Jenkins.

01 Open the **Console Output** and find the first error.
02 **Checkout fails:** check credentials, URL, branch, webhook.
03 **Build fails:** check JDK/Maven version and dependencies; reproduce locally.
04 **Tests fail:** read the JUnit report; look for flaky or environment-dependent tests.
05 **Agent/infra:** check disk space, agent online, permissions.
06 Fix, commit, rebuild; add a test or a check so it cannot recur.

Q20 Justify the role of CI/CD in increasing developer productivity and product stability.

- **Productivity:** no manual builds or deploys; quick feedback; less time on merge conflicts.
- **Stability:** every change tested automatically; small releases are easy to fix or roll back; consistent environments.

Q21 Design a Jenkinsfile script supporting branching strategies and conditional build triggers.

```
pipeline {
  agent any
  stages {
    stage('Build & Test') { steps { sh 'mvn clean verify' } }
    stage('Deploy to Dev') {
      when { branch 'develop' }
      steps { sh './deploy.sh dev' }
    }
    stage('Deploy to Prod') {
      when { branch 'main' }
      steps { input 'Approve production?'; sh './deploy.sh prod' }
    }
    stage('PR checks') {
      when { changeRequest() }
      steps { sh 'mvn sonar:sonar' }
    }
  }
}
```

The **when** conditions run stages only for certain branches or pull requests; feature branches just build and test.

05 Virtualization and Containerization

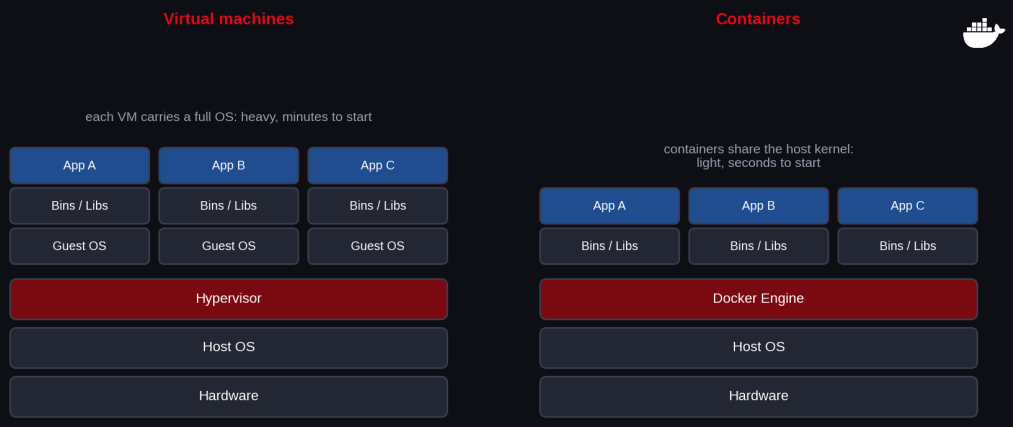
MODULE 5 • DOCKER, DOCKERFILE, CONTAINERS VS VMS

REMEMBER THESE TOOLS

Docker | Kubernetes | Podman | OpenShift | Linux | Java | Tomcat | Nginx

***Memory hook: Dockerfile → build → Image → run → Container**. Containers share the host kernel; VMs carry a full guest OS.*

Virtual machines vs containers



Virtual machines vs containers

Q1 Select configuration parameters used during container creation.

Option	Meaning
--name web	Container name
-d / -it	Detached / interactive terminal
-p 8080:80	Publish port host:container
-v /data:/app/data	Mount a volume
-e KEY=value	Environment variable
--network net1	Attach to a network
--memory 512m --cpus 1	Resource limits
--restart always --rm	Restart policy / auto-remove

Q2 Compare containers and virtual machines based on resource usage.

	Containers	Virtual machines
OS	Share host kernel	Full guest OS each
Size	MBs	GBs
Startup	Seconds	Minutes
Density	Many per host	Few per host
Overhead	Low	High (hypervisor)
Isolation	Process level (weaker)	Hardware level (stronger)

Q3 / Q14 Construct a Dockerfile for deploying a multi-tier web application using custom base images. *(Q3 and Q14 are the same question; answered once.)*

Tiers: Nginx front end, Java backend, MySQL database. First a **custom base image** that every service reuses:

```
# base/Dockerfile  (custom base image: docker build -t myorg/base-jre:1.0 base/)
FROM eclipse-temurin:17-jre-alpine
RUN addgroup -S app && adduser -S app -G app
USER app

# backend/Dockerfile  (multi-stage build)
FROM maven:3.9-eclipse-temurin-17 AS build
WORKDIR /src
COPY pom.xml .
RUN mvn -q dependency:go-offline
COPY src ./src
RUN mvn -q package -DskipTests

FROM myorg/base-jre:1.0
COPY --from=build /src/target/app.jar /app/app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/app/app.jar"]

# frontend/Dockerfile
FROM nginx:alpine
COPY dist/ /usr/share/nginx/html
EXPOSE 80

# docker-compose.yml  (ties the tiers together)
services:
  web:     { build: ./frontend, ports: ['80:80'], depends_on: [api] }
  api:     { build: ./backend,  environment: [DB_HOST=db], depends_on: [db] }
  db:      { image: mysql:8, environment: [MYSQL_ROOT_PASSWORD=secret], volumes: ['dbdata:/var/lib/mysql'] }
volumes: { dbdata: {} }
```

Q4 Analyze resource isolation in containers using Linux namespaces and cgroups.

Mechanism	Role
PID namespace	Container sees only its own processes
NET namespace	Own network interfaces, IP, ports
MNT namespace	Own file-system view
UTS / IPC / USER	Own hostname, inter-process communication, user IDs
cgroups	**Limit and account** CPU, memory, disk I/O so one container cannot starve others

Namespaces = what a container can see; cgroups = how much it can use.

Q5 Match Dockerfile commands with corresponding container build actions.

Command	Build action
FROM	Choose the base image
WORKDIR	Set the working directory
COPY / ADD	Copy files into the image
RUN	Execute a command at build time (new layer)
ENV	Set environment variables
EXPOSE	Document the port
CMD / ENTRYPOINT	Default command when the container starts
USER / VOLUME	Run as a user / declare a mount point

Q6 Demonstrate the steps for creating a custom container using a Dockerfile.

01 Write the **Dockerfile** (FROM, COPY, RUN, CMD).
02 **Build: docker build -t myapp:1.0 .**
03 **Run: docker run -d -p 8080:8080 --name app myapp:1.0**
04 Check: **docker ps**, **docker logs app**.
05 Optionally push: **docker push myorg/myapp:1.0**.

Q7 Identify file types produced during Docker image build processes.

- **Layer archives** (.tar) for each instruction; the **image manifest** (manifest.json) and **image configuration** (JSON).
- **Build cache** and the final tagged **image**; saved images as **.tar** (docker save). Inputs: Dockerfile and .dockerignore.

Q8 Evaluate Docker's role in simplifying software delivery across different environments.

The image bundles code, runtime and dependencies, so **it runs the same on a laptop, test server and production**. This removes "works on my machine", speeds up setup and rollback, and fits CI/CD (build once, deploy anywhere).

Q9 Recommend a container management solution for automating lifecycle operations across OS variants.

Kubernetes with a container runtime (containerd). It automates deploy, scale, update and self-healing across Linux (and Windows) nodes; use **Docker Desktop / Podman** for developer machines.

Q10 / Q21 Outline the tasks involved in deploying a Java web application using Docker. Develop a stepwise deployment plan for running Java services inside Docker containers.

01 Build the app: **mvn clean package** gives a WAR/JAR.
02 Write a **Dockerfile** (base JRE image, copy jar, EXPOSE, ENTRYPOINT).
03 **docker build** and test locally with **docker run**.
04 Tag and **push to a registry** (Docker Hub / private).
05 On the server **pull and run** with ports, env variables, volumes and restart policy (or Compose / Kubernetes).

06 Add health checks, logging and monitoring; automate all steps in Jenkins.

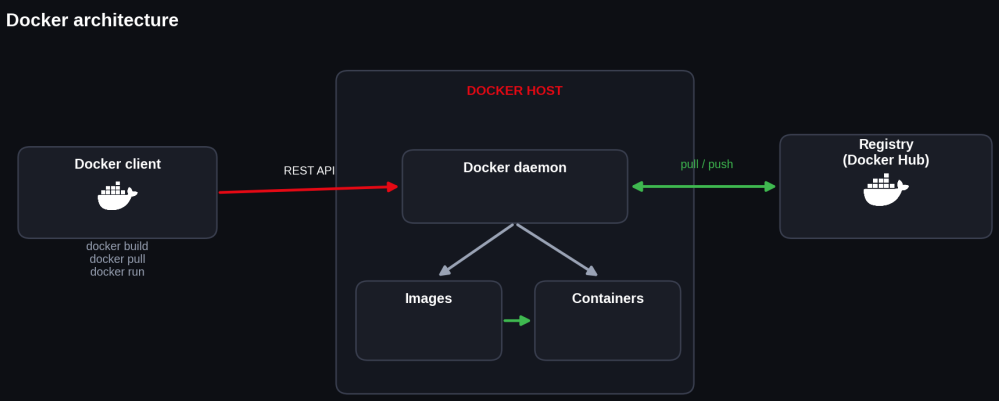
```
FROM eclipse-temurin:17-jre
COPY target/shop.jar /app/shop.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/app/shop.jar"]
```

Q11 Identify two benefits that containerization offers in DevOps practices.

- **Consistency / portability** across dev, test and production.
- **Speed and efficiency:** start in seconds, high density, easy scaling and fast CI/CD.

Q12 List components involved in the container lifecycle.

Dockerfile, image, container, Docker daemon (engine), registry, volumes, networks. States: created → running → paused → stopped → removed.



Docker architecture: client, daemon, images, containers, registry

Q13 Describe the process of container instantiation using Docker CLI.

01 **docker pull nginx** downloads the image from the registry (if not present).
02 **docker create** makes the container (writable layer on top of the image).
03 **docker start** runs it; **docker run** does pull + create + start in one step.
04 **docker ps / logs / exec** manage it; **docker stop**, **docker rm** end it.

```
docker run -d --name web -p 8080:80 nginx
```

Q15 Distinguish features of Docker images and containers with practical scenarios.

	Image	Container
Nature	Read-only template (layers)	Running instance with a writable layer
State	Static, stored in registry	Dynamic, has processes and data
Count	One image	Many containers from it
Scenario	Build **shop:1.0** once in CI	Run 5 copies of shop:1.0 to handle traffic

Q16 Compare Docker with legacy deployment methods using parameters like overhead, portability, and setup.

	Legacy (manual / VM)	Docker
Overhead	High (full OS)	Low
Portability	Low, environment differences	High, same image everywhere
Setup	Hours, manual install	Minutes, one command
Scaling / rollback	Slow	Fast

Q17 Analyze performance differences observed between containerized and virtualized systems.

- **Containers:** near-native CPU and I/O, start in seconds, little memory overhead, more instances per host.
- **VMs:** hypervisor and guest OS overhead, slower boot, more memory; but stronger isolation and can run different OS kernels.

Q18 Choose appropriate Docker instructions to optimize image layering.

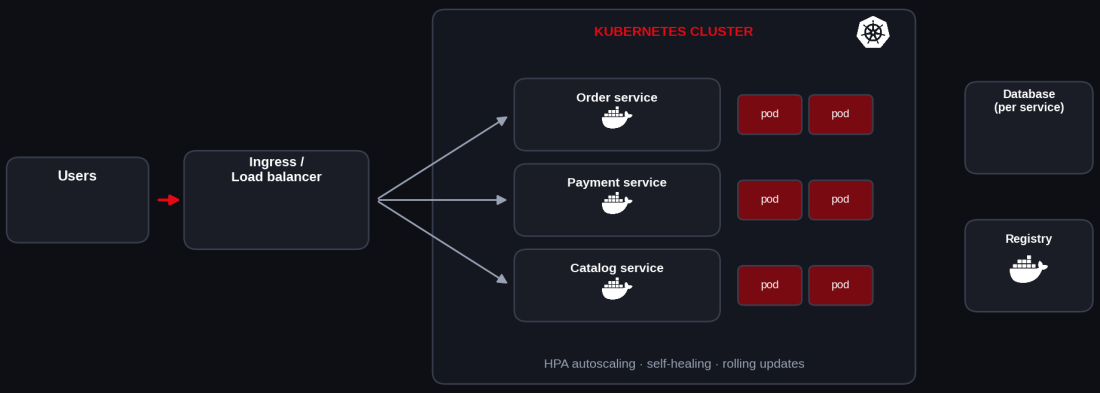
- Put rarely changing steps first (**FROM, dependencies**) and frequently changing **COPY src** last to use the cache.
- **Combine RUN commands** with && and clean caches in the same layer.
- Use **multi-stage builds**, small bases (alpine), and a **.dockerignore** file.

Q19 Name two Docker platforms used in modern infrastructure.

Docker Engine (with Docker Desktop for developers) and **Docker Hub** (registry); also **Docker Swarm** for orchestration.

Q20 Design a scalable container-based architecture for microservice-based application deployment.

Scalable container architecture for microservices



Microservices on Kubernetes

- Each microservice is its own image and Deployment; an **Ingress / load balancer** routes traffic.
- **Horizontal autoscaling**, self-healing and rolling updates; a database per service; images from a private registry.
- CI/CD builds and deploys each service independently; Prometheus and Grafana monitor.

Q22 Examine roles of container registries in continuous deployment pipelines.

- **Store and version images** (tags) built by CI; act as the hand-over between build and deploy.
- Clusters **pull** images to deploy; support **vulnerability scanning, signing and access control**.
- Enable **promotion** (dev → staging → prod) and quick rollback to an older tag. Examples: Docker Hub, ECR, GCR, Harbor.

06 Continuous Testing and Configuration Management

MODULE 6 • SELENIUM, ANSIBLE, CHEF, PUPPET

REMEMBER THESE TOOLS

Selenium | JUnit | Jenkins | Ansible | Chef | Puppet | Terraform | SonarQube

> ***Memory hook: Ansible** = agentless, YAML, push. **Puppet** = agent, declarative, pull. **Chef** = agent, Ruby recipes, pull. Selenium tests the **UI**.*



Continuous testing loop

Q1 List two phases involved in the agile testing life cycle.

Agile test planning and **test execution within the sprint** (continuous). The full cycle: impact assessment, agile test planning, daily stand-ups and execution, release readiness, test agility review.

Q2 Identify two tools commonly used for software configuration management.

CONFIGURATION MANAGEMENT

Ansible | Puppet | Chef | Terraform

Q3 Mention commands used to execute a test case in Selenium.

```
driver = webdriver.Chrome()
driver.get('https://example.com/login')          # open page
driver.find_element(By.ID, 'user').send_keys('asha')
driver.find_element(By.ID, 'pass').send_keys('1234')
driver.find_element(By.ID, 'login').click()        # click
assert 'Welcome' in driver.title                   # verify
driver.quit()                                      # close
# run:  pytest test_login.py     or    mvn test (Java)
```

Q4 Match provisioning tools (Chef, Ansible, Puppet) with their configuration styles.

Tool	Language / style	Model
Chef	Ruby recipes and cookbooks; procedural (imperative)	Agent (chef-client), pull
Puppet	Puppet DSL manifests; declarative (desired state)	Agent, pull from master
Ansible	YAML playbooks; simple, ordered tasks	Agentless over SSH, push



Puppet and Chef (agent based, pull)

Q5 Select appropriate testing approach for a continuously integrated web application.

Automated test pyramid run on every commit: many fast **unit tests** (JUnit), fewer **API/integration tests**, a small set of **Selenium UI tests**, plus regression, performance and security checks, all in the Jenkins pipeline.

Q6 Point out provisioning script components in a Chef recipe.

- **Resources** (package, service, file, template, directory), **attributes** (variables), **cookbook** and recipe files, **run list**, metadata, templates.
- **Notifications / guards** (only_if, not_if) make recipes idempotent.

```
package 'apache2' do
  action :install
end
service 'apache2' do
  action [:enable, :start]
end
file '/var/www/html/index.html' do
  content '<h1>Hello DevOps</h1>'
end
```

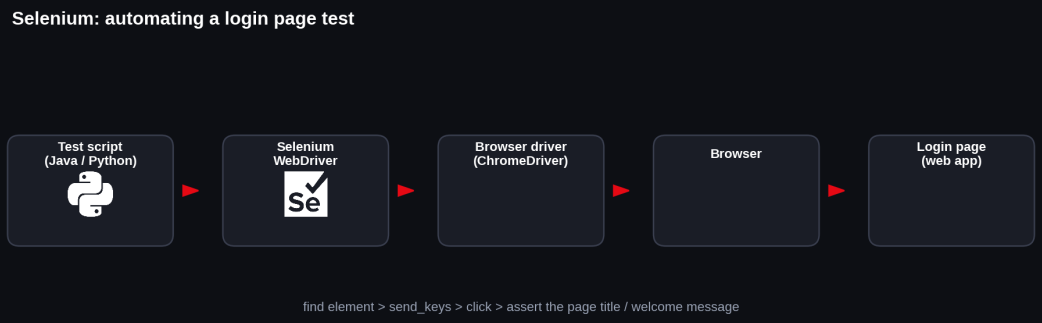
Q7 Classify configuration management tools based on agent/agentless operation.

Agent-based	Agentless
Puppet, Chef, SaltStack (minion)	Ansible (SSH), Salt-SSH, Terraform (API based)

Q8 Explain how continuous testing integrates with DevOps feedback loops.

Automated tests run at **every stage** of the pipeline (commit, build, deploy, release) and report pass/fail immediately to developers. Results from production monitoring become **new test cases**. This shortens feedback time, finds defects when they are cheap to fix and gives a risk score for each release.

Q9 Analyze the execution flow of Selenium in automating UI testing for a login page.



Selenium execution flow

01 Test script starts a **WebDriver** session; the browser driver launches the browser.
02 **Locate** elements (ID, name, XPath, CSS).
03 **Act:** send_keys, click.
04 **Assert** the result (page title, welcome text, error message for bad password).
05 **Report** and close the browser; Jenkins collects results.

Q10 Illustrate the structure and features of a Puppet manifest with an example.

A manifest (.pp) declares the **desired state** using **resources** (package, service, file, user), grouped in **classes**. Features: declarative, idempotent, dependency ordering (require, notify), facts and variables.

```
class webserver {
  package { 'apache2': ensure => installed }
  service { 'apache2':
    ensure  => running,
    enable  => true,
    require => Package['apache2'],
  }
  file { '/var/www/html/index.html':
    content => 'Hello DevOps',
    notify  => Service['apache2'],
  }
}
node 'web01' { include webserver }
```

Q11 Evaluate differences in configuration approaches of Ansible and Chef with real cases.

	Ansible	Chef
Language	YAML playbooks	Ruby DSL recipes
Agent	None (SSH)	chef-client on every node
Learning curve	Easy	Steeper (Ruby)
Model	Push	Pull
Real case	Quick app deployments and cloud provisioning by small ops teams	Large enterprise fleets with complex, long-lived server policies

Q12 Demonstrate test automation strategy for a continuously deployed e-commerce platform.

Level	What is tested	Tool
Unit	Cart, price calculation	JUnit
API	Orders, payments, inventory	Postman / REST Assured
UI	Search, add to cart, checkout	Selenium
Performance	Sale-day load	JMeter
Production	Smoke tests, canary, monitoring	Prometheus, synthetic checks

Run unit and API tests on every commit, UI tests on each deployed build, performance nightly; failures block the release.

Q13 Compare manual testing life cycle with continuous testing practices in agile delivery.

	Manual testing	Continuous testing
When	After development, at the end	At every commit, throughout
Execution	Human, repetitive	Automated, parallel
Feedback	Days	Minutes
Coverage / regression	Limited	Broad, repeatable
Release speed	Slow	Fast

Q14 Construct a use case showcasing end-to-end provisioning using Ansible.



Ansible control node and managed nodes

Use case: provision three web servers for a new application.

```
# inventory.ini
[web]
10.0.0.11
10.0.0.12
10.0.0.13

# site.yml
- hosts: web
  become: yes
  tasks:
    - name: Install Nginx
      apt: { name: nginx, state: present }
    - name: Copy site
      copy: { src: index.html, dest: /var/www/html/index.html }
    - name: Start Nginx
      service: { name: nginx, state: started, enabled: yes }
# run:  ansible-playbook -i inventory.ini site.yml
```

Q15 Design a test automation framework using Selenium for a multi-form web application.

- **Page Object Model:** one class per page/form (LoginPage, AddressPage, PaymentPage) holding locators and actions.
- **Test runner:** JUnit/TestNG (Java) or pytest; **data-driven** tests from CSV/Excel/JSON for the form inputs.
- **Utilities:** waits, config file, screenshots on failure, logging; **reports** (Allure / Extent).
- **Execution:** Selenium Grid / Docker for cross-browser, triggered by Jenkins on each build.

Q16 Evaluate the effectiveness of configuration management tools in reducing deployment risks.

- **Idempotent, repeatable** configuration removes manual mistakes and drift.
- Identical dev, test and prod environments reduce “works only on test” failures.
- Version-controlled configs give audit, review and quick rollback; automated compliance checks cut risk further.

Q17 Create a testing life cycle model incorporating automation tools for continuous feedback.

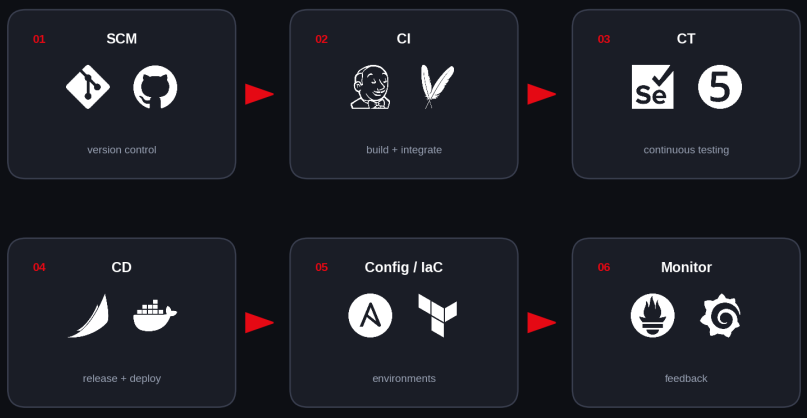
Plan tests (Jira) → write automated tests (JUnit, Selenium) → run on commit (Jenkins) → report and quality gate (SonarQube) → fix and re-run (Git) → monitor production and add new tests. See the testing loop figure at the start of this module.

Q18 Analyze the provisioning pipeline using Chef in a scalable multi-node deployment environment.

01 **Workstation:** write cookbooks and recipes; test them (Test Kitchen); upload with **knife**.
02 **Chef server:** stores cookbooks, roles, run lists and node data.
03 **Bootstrap nodes: knife bootstrap** installs chef-client and registers the node.
04 **chef-client** on each node periodically pulls its run list, converges to the desired state and reports back.
05 Scale by adding nodes with the same role; roles and environments keep configuration consistent.

Q19 Recommend a toolchain combining testing, provisioning, and configuration management in DevOps.

Git (code and configs) → **Jenkins** (CI/CD) → **JUnit + Selenium + SonarQube** (testing) → **Terraform** (provision cloud resources) → **Ansible** (configure servers) → **Docker + Kubernetes** (run) → **Prometheus + Grafana** (monitor).



Recommended toolchain

Q20 Build a configuration-as-code setup using Ansible to provision a LAMP stack for production deployment.

```
- hosts: production
  become: yes
  vars: { db_password: "{{ vault_db_password }}" }   # from ansible-vault
  tasks:
    - name: Install Apache, PHP, MySQL
      apt: { name: [apache2, php, libapache2-mod-php, mysql-server, python3-pymysql], state: present, update_
    - name: Start services
      service: { name: "{{ item }}", state: started, enabled: yes }
      loop: [apache2, mysql]
    - name: Create database
      mysql_db: { name: appdb, state: present }
    - name: Deploy application
      copy: { src: app/, dest: /var/www/html/ }
      notify: restart apache
  handlers:
    - name: restart apache
      service: { name: apache2, state: restarted }
```

Keep this playbook, the inventory and roles in **Git**; Jenkins runs it; secrets stay in **Ansible Vault**.

Q21 Justify the transition from traditional configuration scripting to tool-based provisioning models.

Traditional scripts	Tool-based (Ansible, Terraform, Chef, Puppet)
Imperative shell scripts, hard to maintain	Declarative, readable, reusable roles and modules
Not idempotent: re-running can break things	Idempotent: safe to run repeatedly
Differ per server, drift over time	Consistent and enforced state
Hard to scale and audit	Scales to thousands of nodes; versioned and auditable