DevOps

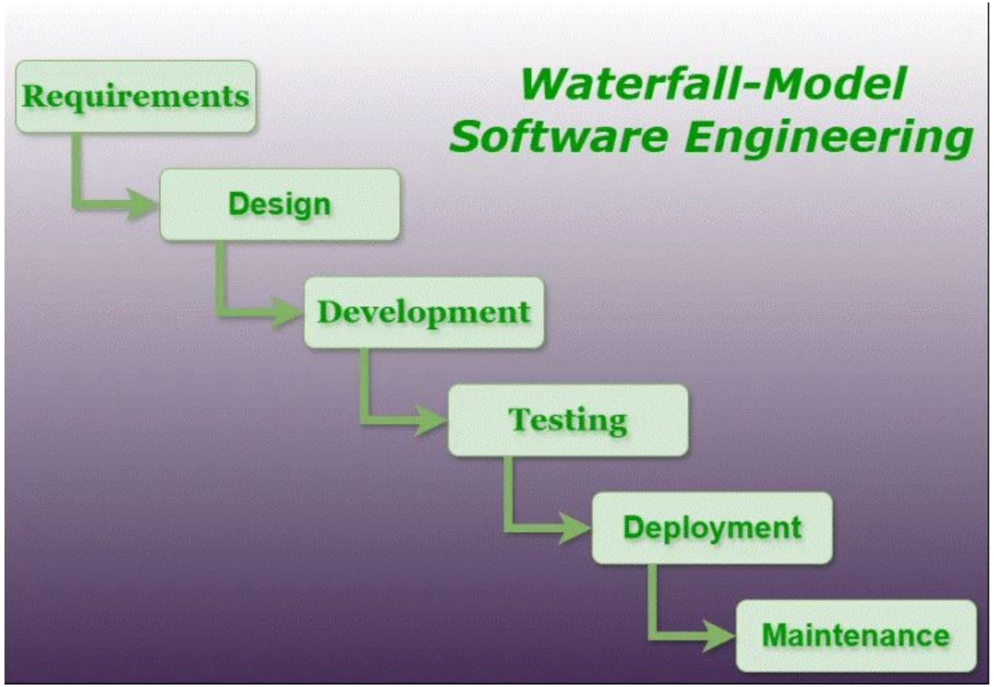
Past Paper Q&A

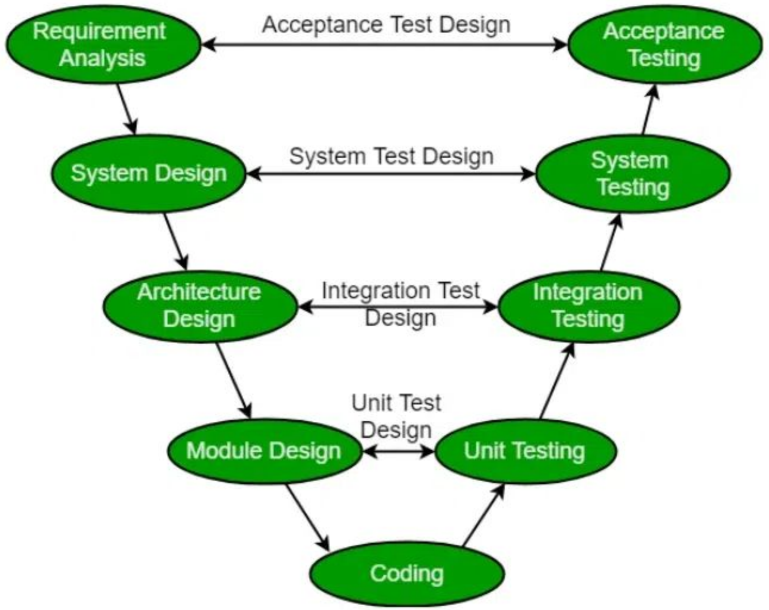
Q1. Your organization is exploring alternative process models to improve the Software Development Life Cycle (SDLC) for cloud-native applications. You have been tasked to analyze these models and suggest improvements. Answer the following questions:

A. Compare the different stages of the SDLC in the V-Model with respect to the waterfall model and explain where they differ? **[3 Marks]**

A. Both the **Waterfall Model** and the **V-Model** follow a sequential order, but the V-Model strengthens validation by pairing every development phase with a corresponding testing phase.

- In the **Waterfall Model**, phases such as requirements, design, coding, and testing occur one after another. Testing begins only after the complete implementation, so defects are often found late.
- In the **V-Model**, the left side of the “V” (requirements, design, coding) mirrors the right side (acceptance, system, integration, and unit testing). Testing is planned in parallel with development.
- The V-Model focuses on *verification and validation at each stage*, enabling early defect detection and reducing rework, unlike the purely sequential nature of the Waterfall Model.


Waterfall-Model
Software Engineering
Requirements
Design
Development
Testing
Deployment
Maintenance


Requirement Analysis
Acceptance Test Design
Acceptance Testing
System Design
System Test Design
System Testing
Architecture Design
Integration Test Design
Integration Testing
Module Design
Unit Test Design
Unit Testing
Coding

B. Your project manager identifies a significant risk in the early stages of the project. Explain how the Spiral Model addresses risk management and how this would benefit the project. **[2 Marks]**

C. How does the Iterative Model help in accommodating changes or enhancements during the SDLC? **[1 Mark]**

B. The **Spiral Model** explicitly incorporates **risk analysis** in every cycle (spiral). Each iteration consists of four phases: planning, risk analysis, engineering, and evaluation. At the risk analysis stage, potential issues—technical, cost-related, or schedule-based—are identified, prototypes may be created to test solutions, and mitigation steps are built into the next iteration.
By continuously identifying and controlling risks early, the project avoids late surprises, reduces failure probability, and improves stakeholder confidence.

C. The **Iterative Model** develops software in repeated cycles, allowing feedback and refinement after each iteration. Each release adds improvements or new features. Hence, it naturally accommodates **changes and enhancements** between iterations without restarting the entire SDLC.

Q2. You are working on a git project with master branch named ***main*** created in the remote repository and shared with the team. It currently contains three files: **input.csv**, **main.py**, and **output.log**. A team member accidentally committed the output.log file. Your team follows FDD and you are required to create a feature branch named ***branch-feature*** for your work. With this context, answer the following questions:

A. You want to normalize the data read from input.csv before further processing and modify the main.py to add function to normalize. Note that input.csv is not to be modified. You also want to correct the above mistake done by team mate and remove output.log file and ensure that log files are not checked in from the current directory or any subdirectories in the future. Provide the necessary git commands to achieve this on the feature branch branch-feature? **[2 Marks]**

```
# Create and switch to new feature branch
git checkout -b branch-feature

# Remove the accidentally committed file
git rm --cached output.log
git commit -m "Remove output.log from version control"

# Add rule to ignore future log files
echo "*.log" >> .gitignore
git add .gitignore
git commit -m "Add .gitignore to ignore all log files"

# Modify main.py to add normalization function
git add main.py
git commit -m "Add normalization function in main.py"
```

B. You have now committed the changes made so far to feature branch. How many blob, tree, and commit objects would be created after performing the above tasks? Justify your answer **[2 Marks]**

When you commit in Git, three types of objects are created — **blobs**, **trees**, and **commits**.

A **blob** stores the actual file content. Since two files were changed (`main.py` modified and `.gitignore` newly added), Git creates **two new blob objects** for these files.

A **tree** object records the directory structure — which files exist and which blobs they point to. So, Git creates **one tree object** for the current state of the project.

Finally, a **commit** object stores metadata such as the author, timestamp, message, and a reference to the tree (and previous commit). Therefore, **one commit object** is created.

In total: **2 blobs + 1 tree + 1 commit = 4 Git objects.**

For a **single Git commit**, Git typically creates:

- **One blob** for *each file whose content actually changed* (new, modified, or deleted files cause new blobs or tree updates).
- **One tree** representing the directory structure of that commit.
- **One commit** object containing metadata and a pointer to that tree (and its parent commit).

C. What type of merge (fast-forward or 3-way) would be used when merging branch-normalize into main assuming no other commits were made to main after creating your feature branch? Justify your answer. **[2 Marks]**

If no new commits were added to *`main`* after the branch was created, the histories are linear. Hence, Git performs a **fast-forward merge**, simply advancing the `main` pointer to the latest commit on `branch-feature`, without creating a new merge commit. A fast-forward merge simply moves the main branch pointer forward to the latest commit on the feature branch, without creating a new merge commit. No changes need to be merged manually, because the two histories are already aligned.

```
Before merge:

main:    A---B

feature: A---B---C---D

After merge (fast-forward):

main:    A---B---C---D
```

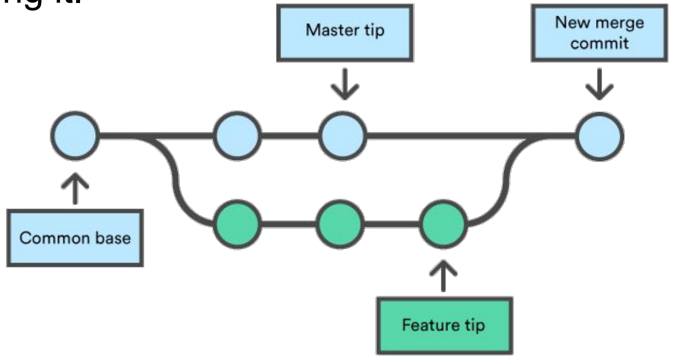
Here, commits C and D are new commits from the feature branch. The main pointer is simply advanced to D — no new commit is created, and the history remains perfectly linear.

However, if new commits were added to **`main`** after creating the feature branch, the histories would have diverged. Now, **`main`** and `branch-feature` each have their own new commits that share a common ancestor but have evolved independently. In this case, Git performs a **3-way merge**.

A **3-way merge** compares three points in history:

1. The **common ancestor** commit (where the branches diverged),
2. The **latest commit on the main branch**, and
3. The **latest commit on the feature branch**.

Git then combines the changes from both sides into a new **merge commit**, which becomes the new tip of `main`. This commit has two parents — one from `main` and one from `branch-feature`. If there are conflicting changes in the same lines of code, Git will pause the merge and ask you to resolve those conflicts manually before completing it.



Q3. Identify and justify whether the following statements are True or False.

A. The CMD instruction in a Dockerfile can be overridden at runtime when starting a container. **[1 Mark]**

True.

In Docker, the `CMD` instruction defines the default command that runs when a container starts. However, it is not fixed — you can override it by specifying another command at runtime using `docker run <image> <new_command>`. This flexibility allows developers to reuse the same image for different purposes, such as testing or debugging, without modifying the Dockerfile itself.

B. Docker containers are operating system independent. **[1 Mark]**

False.

Although containers provide isolation similar to virtual machines, they rely on the host operating system's kernel. For example, a Linux container can only run on a Linux host (or a compatible virtualized Linux environment on Windows or macOS). Containers package the application and dependencies, but not the OS kernel, making them portable only across systems with the same kernel family — not entirely OS-independent.

C. Can we have the two docker containers running on the same host port? **[1 Mark]**

False.

Each host port can be bound to only one container process at a time. If you attempt to run two containers using the same host port (for example, port 8080), Docker will throw an error because of a port binding conflict. To resolve this, containers must either use different host ports or be run on separate network interfaces or bridge networks, ensuring each has a unique mapping to avoid collisions.

D. In Git, a blob object is created for every commit, even if no file content has changed. **[1 Mark]**

False.

Git is designed to be efficient. When you make a commit, Git checks whether the content of a file has changed. If not, it reuses the existing blob object instead of creating a new one. Only modified files generate new blob objects, while the new commit object references existing blobs and trees. This design makes Git's storage mechanism highly space-efficient.

E. Docker images are immutable. **[1 Mark]**

True.

Once built, a Docker image cannot be changed directly; any modification results in the creation of a new image layer. This immutability ensures consistency and reproducibility across environments — whether you run the container on your laptop, in a testing environment, or in production, the behaviour remains the same. Immutability is a key principle that supports versioning, rollback, and auditability in containerised deployments.

F. Docker containers share the host operating system kernel. **[1 Mark]**

True.

Unlike virtual machines that run their own OS kernels, containers share the kernel of the host system while maintaining isolated user spaces. This sharing is what makes containers lightweight and fast to start. They use kernel-level features like namespaces and cgroups to isolate processes, resources, and file systems. This efficient sharing of the host kernel is central to Docker's low overhead and scalability advantages.

Q4. You are working on a movie ticketing application that was originally built using a monolithic architecture that are all bundled into a single codebase and deployed as a unified entity. As the user base has increased, your team has faced challenges as follows:

Challenge#1: Difficulty in scaling specific components (e.g., ticket booking during peak times),

Challenge#2: Lengthy deployment cycles requiring a complete redeploy for minor updates etc).

To tackle these challenges, your team has decided to transition from a monolithic architecture to a microservices architecture. With this context, answer the following questions:

A. Identify at least 2 key features of the movie ticketing application and discuss how each feature can be implemented in both monolithic and microservices architectures. **[2 Marks]**

A. Two Key Features

1. **User Management**
   - *Monolithic:* User authentication, profiles, and permissions handled in the same large codebase and database.
   - *Microservices:* A dedicated `user-service` manages user data and authentication through REST or gRPC APIs, with its own database for better isolation.
2. **Ticket Booking**
   - *Monolithic:* Booking logic, payment, and seat allocation coded together; any small change requires redeploying the full application.
   - *Microservices:* Separated into independent services such as `booking-service`, `payment-service`, and `notification-service`, communicating asynchronously (e.g., via message queues).

B. Discuss how Challenge#1 stated above can be addressed by transitioning to a microservices architecture.? **[2 Marks]**

In a monolithic application, all components such as user management, payment, and ticket booking are tightly coupled within one deployable unit. This means if the ticket booking feature experiences a surge in demand—for example, during a popular movie release—you are forced to scale the *entire* application, even though only one part actually needs more resources.

In contrast, a **microservices architecture** breaks the system into independent services, each with its own runtime and deployment. This allows **fine-grained, independent scalability**. When the ticket booking workload spikes, only the `booking-service` can be scaled horizontally (for instance, by adding more container replicas or Kubernetes pods) without affecting other services such as `user-service` or `review-service`. This approach uses computing resources efficiently, reduces operational costs, and ensures better responsiveness and system stability during peak traffic.

C. Discuss how Challenge#2 stated above can be addressed by transitioning to a microservices architecture? **[2 Marks]**

In a monolithic system, even a small change—like fixing a bug in the payment module—requires rebuilding, retesting, and redeploying the entire application. This results in long deployment cycles, higher risk of introducing new issues, and frequent downtime.

Microservices eliminate this problem through **independent development and deployment pipelines**. Each service, such as `payment-service`, `notification-service`, or `UI-service`, can be built, tested, and deployed separately using CI/CD tools like Jenkins or GitHub Actions. This means you can update or roll back just one service without touching others. The result is faster delivery of features, reduced downtime, easier rollback in case of errors, and an overall smoother **continuous integration and delivery (CI/CD)** process that supports agile development and high availability.

Q5. You are working on a Gradle-based Java project named WordCounterDemo that includes functionality for counting frequency of words of a given document written in App.java and have written unit test cases in AppTest.java using Junit (version 4.13.2). Additionally, you are using SonarQube for static code analysis. With this context, answer the following questions:

A. What command would you run to trigger a SonarQube analysis of the project? Describe what happens during this process. **[2 Marks]**

The command to run a SonarQube analysis for a Gradle project is:

```
gradle sonarqube
```

When you execute this command, Gradle uses the **SonarQube plugin** to perform a **static code analysis** of your project. The process typically follows these steps:

1. **Compilation and Testing:** Gradle compiles the Java source files (e.g., `App.java` and `AppTest.java`) and runs the JUnit tests using version 4.13.2 to ensure the project builds successfully and all tests pass.
2. **Code Metrics Collection:** The SonarQube plugin then inspects the compiled code and test results, collecting metrics like lines of code, cyclomatic complexity, test coverage, duplication, and potential bugs or code smells.
3. **Upload to SonarQube Server:** These metrics and reports are then sent to the configured SonarQube server, where they are analysed against predefined **quality gates** (thresholds for code quality).
4. **Dashboard Visualisation:** Once the analysis completes, you can view results through the SonarQube web dashboard — including issues categorised as *bugs, vulnerabilities, code smells,* and *maintainability metrics*.

In short, `gradle sonarqube` automates the **entire static code analysis pipeline** — from compilation and testing to quality reporting — ensuring that your Java project meets both functional and non-functional (quality) standards before deployment.

B. Suppose the project depends on an external library like Apache Commons Lang (version 3.12.0). Write the dependency configuration for this library in build.gradle **[1 Mark]**

External Dependency Configuration

In Gradle, dependencies are declared inside the `dependencies` block of the `build.gradle` file.
To include the **Apache Commons Lang** library (version 3.12.0), you would add:

```
dependencies {

    implementation 'org.apache.commons:commons-lang3:3.12.0'

}
```

Here's what happens:

- **`implementation`** specifies that this dependency is required at both compile-time and runtime for your project but is not exposed to downstream modules (modern Gradle best practice).
- Gradle automatically fetches the dependency from the **Maven Central repository** (or whichever repository is configured) and caches it locally for reuse.
- The library provides useful utility classes such as `StringUtils`, `ObjectUtils`, and `ArrayUtils`, which simplify string manipulation and object handling in your Java application.

C. Gradle supports incremental builds, making it faster for repeated builds. State true or false and justify **[1 Mark]**

Gradle's **incremental build mechanism** intelligently determines which parts of your project need to be rebuilt by comparing the **inputs (source files, resources, dependencies)** and **outputs (compiled classes, JARs)** of each task. If nothing has changed since the last build, Gradle **skips redundant tasks** and reuses previously generated results from its build cache.

For example, if you modify only `App.java`, Gradle recompiles only that file and the affected dependencies, not the entire project. This makes subsequent builds dramatically faster — an essential feature for large-scale projects or frequent CI/CD pipelines.

Additionally, Gradle supports **build caching** (both local and remote) to share compiled artifacts across builds and even across machines, further improving performance and consistency.

D. Explain what happens when you run gradle testClasses? Will main/java files be compiled? Justify your answer. **[2 Marks]**

When you run the command:

```
gradle testClasses
```

Gradle executes a built-in task called `testClasses`, which is responsible for compiling all the test files located under `src/test/java`. In a standard Java project, there are two main folders: `src/main/java`, which contains your main source files like `App.java`, and `src/test/java`, which contains your JUnit test files such as `AppTest.java`. Since the test code depends on the main application classes, Gradle must ensure that the main code is compiled first before compiling the test code. To handle this automatically, Gradle uses a **task dependency chain**, where the `testClasses` task depends on another task called `classes`. The `classes` task compiles your main source files and places the resulting `.class` files in the `build/classes/java/main/` directory. Only after this step is complete does Gradle compile the test files, placing their compiled versions in `build/classes/java/test/`.

It is important to note that running `gradle testClasses` does **not** execute any tests; it only compiles them. This task compiles all test source files under `src/test/java` and ensures that both the main application classes (from `src/main/java`) and the test classes are compiled into bytecode and placed in their respective `build/classes` directories.

To actually execute the tests and obtain results such as “passed” or “failed,” you must run:

```
gradle test
```

The `test` task depends on `testClasses`, so it first compiles the necessary source and test files (if not already up to date) and then executes all compiled tests using the configured test framework, such as **JUnit**. The `gradle test` task also generates a detailed test report summarising which tests passed, failed, or were skipped.