DEPLOYMENT PIPELINE

Components of a Deployment Pipeline:

- Build
- Automated Testing
- Staging
- Quality Checks
- Manual Approvals
- Deployment

Tools for Deployment Pipelines:

- Popular CI/CD tools
- Parallel and Sequential Stages
- Rollbacks
- Infrastructure as Code (IaC)(AWS,Terraform)
- Continuous Improvement

Build Tools

Build tools automate software development tasks like compiling code and running tests. They can be **task-oriented** (e.g., Ant, NAnt, MsBuild) or **product-oriented** (e.g., Make). Task-oriented tools focus on tasks, while product-oriented tools focus on the products they generate. Build tools have dependencies and prerequisites, which must be executed correctly for successful builds. Some tools support **incremental builds**, compiling only changed files. In languages with virtual machines, optimization happens during runtime via the Just In Time (JIT) compiler.

Task-oriented tools track each task to ensure it's executed only once, even if referenced multiple times. Product-oriented tools manage outputs like compiled files, using timestamps to determine if recompilation is needed, optimizing the build process.

Build tools :

Make: Tracks dependencies but becomes hard to manage in complex projects. Often replaced by SCons for C++. Susceptible to errors like whitespace issues and OS-specific scripts.

Ant: Java-focused, cross-platform, task-oriented. Uses complex XML scripts, which can be confusing and hard to maintain. Flexible but newer tools like Maven and Gradle are preferred.

NAnt/MSBuild: For .NET, similar to Ant with XML scripts. Integrated into Visual Studio but shares Ant's complexity. Suitable for .NET projects but can be difficult for large builds.

Maven: Simplifies Java builds and manages dependencies. It enforces a strict structure, which can be limiting. Automatic plugin updates may cause unpredictable builds and dependency issues.
Rake: Ruby-based, flexible, using Ruby scripts for tasks. Easier than XML tools but requires Ruby runtime and gem interaction. Suitable for various platforms with good debugging support.

Buildr: Built on Rake, simplifies dependency management for Java projects. Faster and easier than Maven, ideal for new projects. Highly customizable and efficient for multi-project builds.

Psake: Basic of Psake is an internal DSL written in power shell, which supports task oriented dependency network. This is meant for Windows users.

Principles and Practices of Build and Deployment Scripting

- **Create scripts for each stage:** Divide the deployment pipeline into separate scripts for each stage, such as commit and acceptance tests, to enhance organization and manage tasks more effectively.

- **Use appropriate technology:** Deploy applications using suitable tools, like Wsadmin, ensuring effective collaboration among developers, testers, and operations staff for a smooth and consistent deployment process.

- **Same script for all environments:** Implement the same deployment script across different environments while separating configuration data and storing it in version control to maintain consistency during deployments.

- **Use OS packaging tools:** Utilize OS-specific packaging tools, such as Debian or RedHat, to package binaries and static files, streamlining the deployment process and improving efficiency and automation.

- **Idempotent deployment:** Ensure that each deployment process leaves the target environment in a consistent and correct state, using a known baseline environment to maintain reliability regardless of initial conditions.

- **Incrementally evolve deployment:** Gradually develop the deployment system by refining scripts and processes, ensuring both operations and developers use the same tools for deployment across various environments.

Commit Stage

The commit stage marks a project as committed to the version control system and can result in either a successful or failed commit. A successful commit generates binary artifacts and deployable assemblies for testing and release. This stage focuses on minimizing integration time and ensuring code quality, allowing for quicker deployment.
When a change is made in the version control system, the continuous integration (CI) server detects it, checks out the source code, and performs several tasks:

- Running commit tests on the integrated code.
- Creating deployable binaries.
- Analyzing the codebase.
- Generating additional artifacts for later use.

These tasks are automated by build scripts executed by the CI server, and the binaries and reports are stored in a central artifact repository.

Commit stage principles and practices.

Provide fast and useful feedback
Commit test failures can occur due to syntax errors, semantic errors, or configuration issues. Developers should be notified quickly with a summary of these failures, making it easier to fix them early.
What should break the commit stage?
The commit stage should fail for compilation issues, test failures, or environmental problems. It needs to provide more detailed metrics, like code coverage. This information can be aggregated in graph or as a sliding scale, rather than just a pass or fail status.
Tend the commit stage carefully
Build scripts and testing tools in the commit stage must be maintained as carefully as the codebase. A poor build system can lead to costly development delays and inefficiencies. constant work needs to be done to improve the quality design and performance of this script in the commit stage.
Give developers ownership
All team members should feel responsible for the commit stage. Experts can help establish best practices and transfer knowledge, fostering a sense of ownership among developers.
Use of a build master for very large teams
For large, distributed teams, a build master can oversee the build process, ensuring discipline and effective collaboration, especially for teams new to continuous integration.

Results of Commit Stage

The commit stage produces outputs like binaries and reports based on the input source code. If tests fail, the generated reports include metrics such as test coverage and complexity, helping to analyze code quality.
Artifact Repository
Outputs from the commit stage are stored in an artifact repository, which is separate from version control systems. This repository tracks artifacts and allows retrieval of any version related to previous software releases without storing failed stages.
Acceptance Criteria for Configuration Management
A good configuration management strategy requires that the binary creation process be repeatable. Modern continuous integration servers provide artifact repositories that manage and store necessary artifacts, allowing for easy access and maintenance.

Role of artifact repository diagram.

1) A team member in the delivery team commits a change.
2) The continuation continues integration server runs the commit stage.
3) On successful completion, the binaries as well as any reports and metadata are saved to the artifact repository.
4) The continuation in continuous integration server, the deployment pipeline retrieves the binaries created by the commit stage and deploys to a production like test and environment.

5) The continuous integration server runs the ‘acceptance test’ and reuse the binaries created by the commit stage.
6) On the successful completion of the acceptance test, the release candidate is marked as having passed the ‘acceptance test’.
7) The testers obtain the list of all binaries which have passed acceptance test and the press of a button runs automated process to deploy them into manual testing environment.
8) Next, Testers perform ‘manual testing’.
9) On successful completion of manual Testing, testers update the status of the release candidate to indicate. It has ‘passed manual testing’.
10) Next, the continuous integration server retrieves, the latest candidate that has passed acceptance, testing or manual, testing depending upon the stage in the pipeline configuration from the artifact repository and deploys. The application to the production test environment.
11) Next, the capacity tests are run against the release can rate.
12) On successful completion of the capacity test, the status of the candidate is updated to “capacity tested “.
13) This is repeated for as many stages in the pipeline.
14) Once the code has passed all the relevant stages, it is ready for release. It can be released by anybody with appropriate authorization. 15) And the conclusion of the release process, the RC is marked as **“released”.**

Design Strategies

Avoid the User Interface : Don’t focus too much on testing the user interface (UI) because it has many moving parts. This can make tests complicated and slow to set up. Instead, test the core features directly for faster results.

Dependency Injection Method : Dependency injection is a way to manage how different parts of your code interact without hardcoding their connections. This makes it easier to change or replace parts of the code, leading to cleaner and easier-to-manage applications.

Avoid the Database : Testing with a database can slow things down because it takes time to set up and clean up the database. By separating your code from the database, you can make tests faster and less complicated.

Avoid Asynchrony in Unit Tests : Testing code that runs tasks at different times (asynchronously) can be tricky. It’s better to split these tests into simple steps, checking each part one at a time to keep things clear and reliable.

Using Test Doubles : Unit tests often need to work with other parts of the code, but this can complicate things. By using test doubles like stubs or mocks, you can pretend to use those parts without needing them to work. This speeds up testing and makes it simpler.

Minimizing State in Tests : Keeping tests simple and focused on what’s necessary is key. Complex setups can confuse you and make tests harder to maintain. Always try to minimize how much setup is needed for a test.

Faking Time : When tests depend on time, it can be hard to manage. By creating a separate class to handle time, you can control how time behaves in tests. This makes the tests more predictable and easier to work with.

Brute Force : Speed is important in testing, but you also need to catch errors effectively. Sometimes it's okay to have a slower test process if it helps find more bugs. Try to keep tests under ten minutes while ensuring they are thorough, and consider using tools to help speed things up.

Note on Version Control System (3 Marks)
A **Version Control System (VCS)** is a tool that helps manage changes to files, documents, or code in a project. It records every change made, allowing users to:

1. **Track Changes**: Keep a history of who made what changes and when.
2. **Collaborate**: Multiple team members can work on the same files simultaneously.
3. **Revert Changes**: Restore files to a previous version if needed.

Examples: Git, SVN, and Mercurial.

Elucidate Configuration Management (3 Marks)

Configuration Management involves managing and tracking all important resources and changes in a project. Its key aspects are:

1. **Versioning**: Stores different versions of files and their relationships.
2. **Change Tracking**: Records who made changes, what was changed, and when.
3. **Reproducibility**: Helps recreate environments and systems for testing or deployment.

It ensures smooth project evolution, compliance, and team coordination.

Configuration Management (Version Control)

Configuration management is the process of managing all the important files, documents, and resources in a project, along with their relationships. It helps keep track of changes made during a project's lifecycle. Each version is uniquely identified using a **build number** and can be updated or modified as needed.
This process ensures that any changes made to the system are properly recorded and can be tracked to show how the system evolved over time.

Key Features of a Good Configuration Management Strategy:

1. **Reproduce Environments**:
   - It should allow creating exact replicas of any setup, including the operating system, network settings, software, and applications.
2. **Make and Deploy Changes**:
   - It should enable smooth updates or incremental changes to individual parts of the system and apply these changes to any or all setups.
3. **Traceability of Changes**:
   - It should track all changes, including details about what changed, who made the change, and when it was made.
4. **Compliance and Satisfaction**:
   - It should meet compliance requirements and ensure the changes meet user needs.
5. **Team Communication**:
   - It should provide updates about changes to all team members to ensure everyone stays informed.

Managing Dependencies

In software projects, managing **dependencies** means handling the external tools, libraries, or modules that your project needs to work. These dependencies can be of two main types:

1. **External Dependencies**:
   - These are third-party libraries or tools (like prebuilt code) that your team uses but doesn't modify.
   - Example: Tools for data processing, UI frameworks, or APIs.
   - These are often downloaded and installed using **package managers** like Ruby Gems or Perl Modules.
2. **Project Components**:
   - Large projects are divided into smaller parts (modules or components) to make development easier.
   - These parts often depend on each other and are frequently updated.
   - Keeping track of these changes and ensuring compatibility is part of dependency management.

How to Manage Dependencies:

- Use package managers to install libraries globally or per project.
- Maintain an **approved library repository** with specific versions to ensure consistency.
- Break big projects into smaller parts to make dependency tracking and updates easie

Managing Software Configuration

Managing software configuration means controlling how the software is set up and works in different environments. The configuration settings decide how the software behaves when it is deployed and running.

Important Points

1. **Keeping Things Consistent**:
   - The configuration team sets up the software in a way that all parts work well together.
2. **Balancing Simplicity and Flexibility**:

    - Allowing too many options for configuration can make the system confusing and harder to manage.
    - It's better to keep things simple at first and add more options later only if needed.
3. **When Configuration is Used**:
   Configuration settings can be applied at different stages:
    - **Build Time**: Settings are added when the software is being created.
    - **Packaging Time**: Settings are added to packaged files (e.g., zip or setup files).
    - **Deployment Time**: Settings are given when the software is installed.
    - **Runtime**: The software loads settings when it starts or while it is running.
4. **Tracking Changes**:
    - Keep a record of any configuration changes between deployments to ensure the software runs as expected.

Why It's Important

Good configuration management helps the software run smoothly, reduces errors, and makes it easier to update or fix when needed.

Continuous Integration (CI) Explained Simply

Continuous Integration (CI) is a method used in software development to ensure that the software being built is always in a working state. It makes the development process faster, more efficient, and less error-prone.

How It Works:

1. **Frequent Code Changes**:
    - Developers regularly commit (save) their code changes to a shared repository.
    - Every time a change is made, the entire application is automatically rebuilt.
2. **Automated Testing**:
    - After building, a set of automated tests is run to check if everything works properly.
3. **Immediate Problem Fixing**:
    - If the build or tests fail, the team stops their work and fixes the problem right away.
4. **Always Working Software**:
    - The main goal of CI is to ensure the software is always in a functional state.
    - This avoids long delays for testing and integration at the end of the project.

Why It's Useful:

- **Saves Time**: Bugs are caught early and fixed quickly, saving time compared to fixing them later.
- **Reduces Costs**: Early fixes are cheaper and prevent costly delays.
- **Faster Delivery**: Teams can deliver working software more quickly.
- **Fewer Bugs**: CI ensures fewer errors in the final product.

Key Difference from Manual Processes:

- In manual development, testing and integration are often delayed until the end, making it harder and slower to fix issues.
- With CI, integration happens continuously, so problems are caught and resolved immediately.