Staging Test Automation Pipeline

As per my prior experience working with both microservices & monolith services the setup can be different for both types of services to have a much efficient and reliable system. Answers will describe approaches for both types of services.

1. Pipeline Setup

The pipeline can be set up using CI/CD pipeline on staging environment which is ideally replication production env, this will ensure that any issue with deployment can be identified before deployment over production.
There are multiple available to automate deployment on staging env.

Example ● GitHub Actions (I have previous experience working with GA)

- **Create a workflow file** in `.github/workflows/`.
- **Trigger whenever there is push to staging** using `on: push: branches: staging`.
- **Build and deploy** using scripts.
- **Run automated tests** once successful mark build as green
- **Create testing reports.**
- **(Optional) Require manual approval** before deploying to production.

- GitLab CI/CD
- Jenkins
- ArgoCD
- Quality enforcement tools (e.g., ESLint, SonarQube) to maintain coding standards.

Set Up the iOS Repository

1. Make sure your **Xcode project** includes a valid **Xcode workspace** (`.xcworkspace`).
2. In your repository, create a `.github` directory.
3. Inside the `.github` directory, create a `workflows/` folder and add a new YAML file to define your pipeline, for example: `yourPreferredName.yml`

Set Up the android Repository

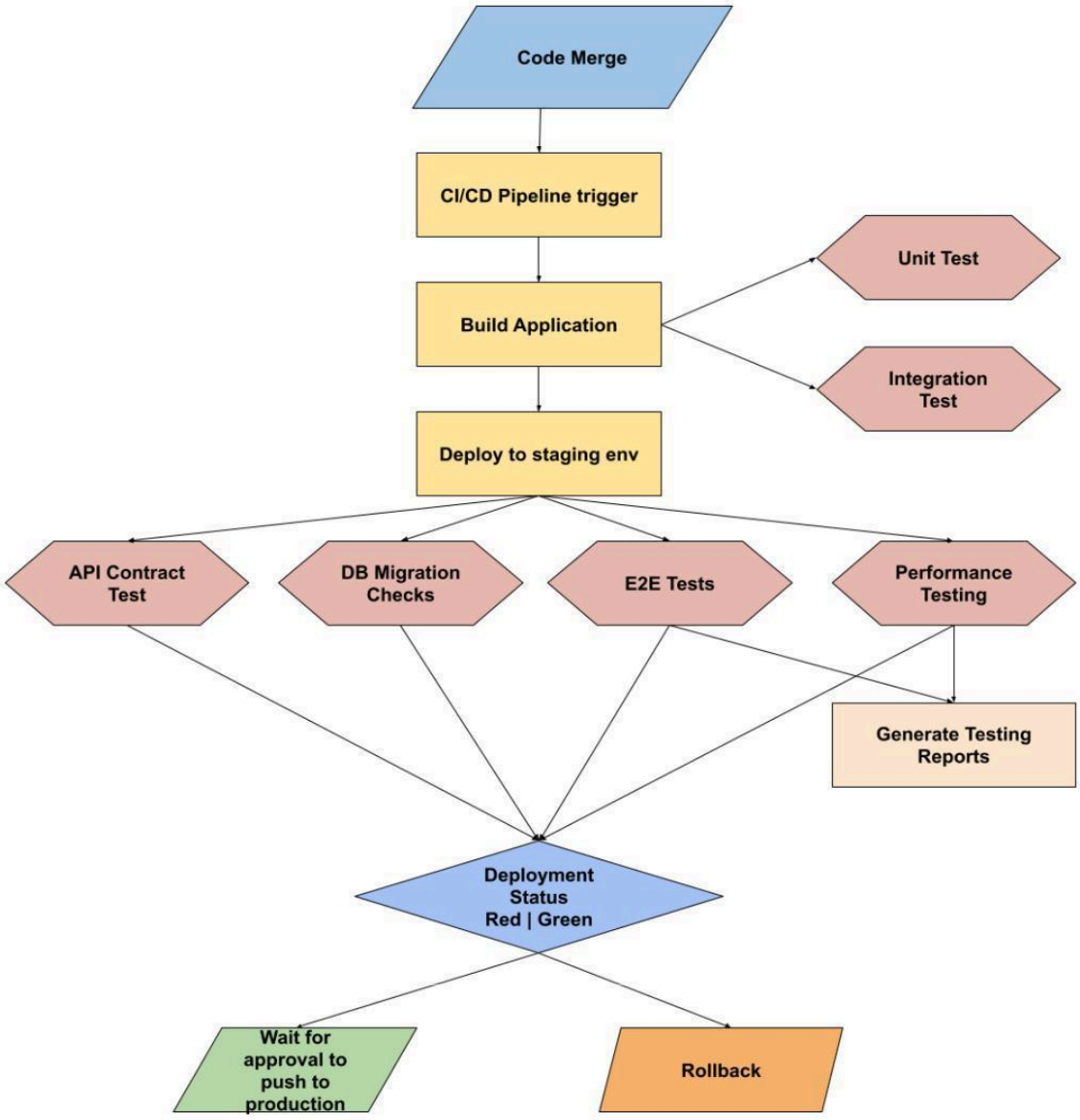
Ensure the android project is using Gradle for building and that you have a valid build `.gradle` file at the root level and module level **`(app/build.gradle).`**

How will the automation pipeline work for Web + Android + iOS?

There are multiple tests to ensure maximum coverage of the application, also these align with the testing pyramid to ensure the tests are faster and efficient.

Test	At what stage will it run	Usage
Unit Tests	During the build phase	Tests individual components and functions. Test iOS components and functions using **XCTest/XCUITest** Test Android components using **JUnit & Mockito**
Integration Tests	During the build phase	Using frameworks like wiremock to test integration with external applications, db.
API Contract Tests	After deployment in staging	To validate if the API correctly follows the contract
Database Migration Tests	After deployment in staging	Checks if database schema changes apply correctly without breaking data
End-to-End (E2E) Tests	After deployment in staging	Tests application complete flow simulating a complete user journey for different flows. Simulate full user journeys in **iOS** app and **Android** app.
Performance & Load **Testing**	After deployment in staging	Simulates high user traffic to check system stability and response time. **For ios** : Xcode Instruments (Official Apple Performance Tool) **For android** : Android Profiler (Built into Android Studio)
Unit Test	After deployment in staging	Runs functional tests on actual UI (e.g., Cypress, Selenium, **iOS** : Functional UI tests using **Appium & XCUITes**t. **Android** : Functional UI tests using **Appium & Espresso**

Below is a simplified flowchart for automation pipeline including multiple tests at various stages.



Monolith vs Microservice

End-to-End (E2E) Testing: In monolith to make the tests run faster and not block the build pipelines for longer it can be helpful to **run only tests associated with part of the code changed**.

Regression Testing: Since microservice are lightweight running full regression tests won't consume much of time so it can be triggered on every build but with monolith running regression tests only specific to component changed can save time.

How to do partial Testing?

- Using Test Impact Analysis (TIA) to run tests only for changed components - this can also be automated using smart test runners in CI/CD.
- Implement tagged test cases to execute relevant test suites.

Example - If there is a change in login component, it will be faster if only tests related to login module are run and not other tests like course/module selection, document upload.

Pipeline trigger frequency

This pipeline will be triggered on every merge/push to staging branch, the only difference in monolith will be that it will not run complete regression suite everytime as that might slow down build pipeline but only impact components tests. In addition also for monolith there will be scheduled pipeline to run complete tests suite for all the components which will not block the build pipeline.

MONOLITH

When?	Action
PR merge to staging branch OR Direct push to staging branch (not recommended for teams)	Run CI/CD with below tests: ● All Unit Tests ● All Integration Tests Changed Component specific : ● API Contract Tests ● End-to-End (E2E) Tests ● UI & Functional Tests ● Performance & Load Tests
Every 6 hours	Run complete test suite including coverage for all the components

MICROSERVICE

When?	Action
PR merge to staging branch OR Direct push to staging branch (not recommended for teams)	Run complete test suite including coverage for all the components

Other application of the automated pipeline

Supporting Large-Scale Codebase Refactoring

Refactoring a big codebase can introduce unexpected issues. The pipeline helps by:

- Automatically running tests after every commit to detect breaking changes early.
- Using Test Impact Analysis (TIA) to only execute tests relevant to modified code, instead of running the entire test suite.
  Fastlane combined with **XCTest** or **XCUITest** to execute only the tests relevant to modified code. This reduces the time required to execute the entire test suite.
- Running API contract tests to ensure that backend changes don't break frontend interactions.
- Checking code quality using tools like SonarQube to detect duplicate code, security flaws, and bad practices.

Example:

Frontend Architecture Migration

Migrating from AngularJS to React or from Class Components to Functional Components in React.

- The pipeline ensures all UI functionality is same after migration.
- Regression tests insures users real time flow works seamlessaly and same as before.

Upgrading Legacy Dependencies

Upgrading from Node.js 16 to Node.js 18

- Uses static code analysis to flag outdated syntax.
- Regression tests insure the existing pipeline does not break after version upgrade.

Improving Application Performance

Refactoring the backend to use Redis caching or optimizing database queries.

- Performance testing ensures that there is no additional lag after migration.
- Regression and UI Testing to ensure the migration does not break existing user flows.

Help developers

With automated pipeline developers can be ensured the latest changes won't break existing flow and also the new flow. Also this pipeline can help developers by:

- Automated tests can identify issues faster than manual testing each flow.
- Code analysis can help by blocking merges that are incorrectly formatted, probable version conflicts.
- Running the tests in parallel can reduce build time.
- In case the developer creates a merge which is breaking existing flow, it can be identified during testing phase in pipeline and the reportings can help with identifying the point of failure.
- Saving time by catching issues early in staging and therefore developers won't have to debug and rollback changes on production, this also helps with good user experience.

Example:

Fast Feedback on Code Changes

A developer pushes code, and within minutes, the pipeline runs unit tests and reports back if the changes break anything

- Helps developers **fix issues early** before
- merging into production Reduces time spent debugging later in the cycle.

Real-Time Monitoring & Alerts

Pipeline can be integrated with slack to give pipeline status green or red over message, so that team has current overview of the pipeline and check in case of any failure in pipeline

Test Impact Analysis

Instead of running all tests on every commit, pipelines can use Test Impact Analysis (TIA) to execute only relevant tests based on the code changes.

Example

If only the login module changes, only tests specific to login module is run and not specific to course selection or document upload.

- Makes the pipeline run faster and unblocks build quickly

Performance Benchmarking

Pipeline will run performance tests and compare with existing benchmarks to notice any degradation in latency.

Example

Using JMeter after deployment, comparing API response times to detect slowdowns.

- Helps prevent regressions in system performance.
- Ensures the application meets performance expectations before release.

Testing for UI consistency

UI testing to ensure the changes don’t break existing UI layout and functionality for each UI component.

Example

After a change the button click option is not working as expected, with the UI tests this will be identified and block the build to prevent the build from rolling out on production.