

CI/CD Best Practices for your DevOps Journey

August 20 2025

Overview

Agenda

- Beyond DevOps: Software Delivery Management
- Summary of 10 CI/CD best practices
- “Double-Click” on key best practices
- Breaks for Discussion
- Questions and Answers



DevOps Elite Performers

46x

more frequent code deploys

i.e. multiple times per day vs. once a week or less

2555x

faster lead-time from commit to deploy

i.e. less than an hour vs. more than a week

2406x

faster time to recover from downtime

i.e. less than an hour vs. weeks

1/7th

as likely that changes will fail

i.e. fail 0-15% of the time vs. 46-60%

But...

Things aren't perfect.

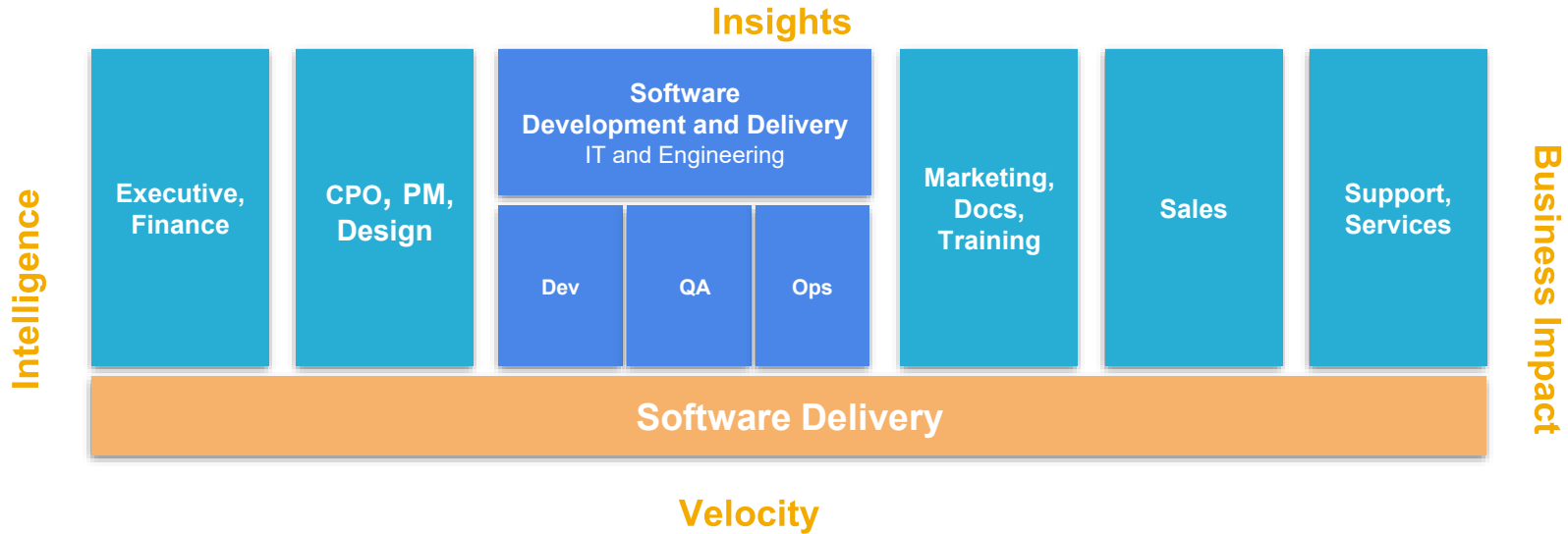


We still have...

Disconnected and Fragmented teams, tools and process

- Disparate, stand-alone software tools
- No common language, data or process sets
- No clear way to ensure we deliver the right thing

It Takes a Village to Truly Deliver Value



Introducing Software Delivery Management

All functions collaborating



Exec/ Finance



CPO/Product
Mgmt/ Design



Dev/QA



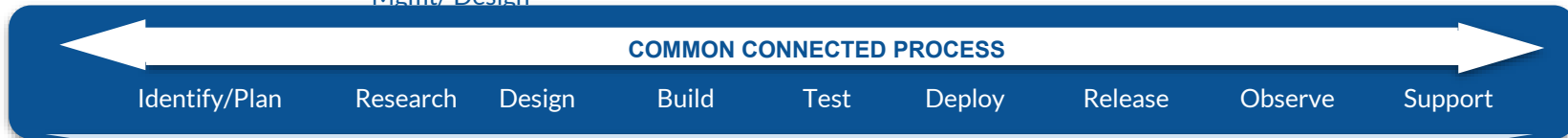
Security/Ops



Sales/Mktg



Support/Services



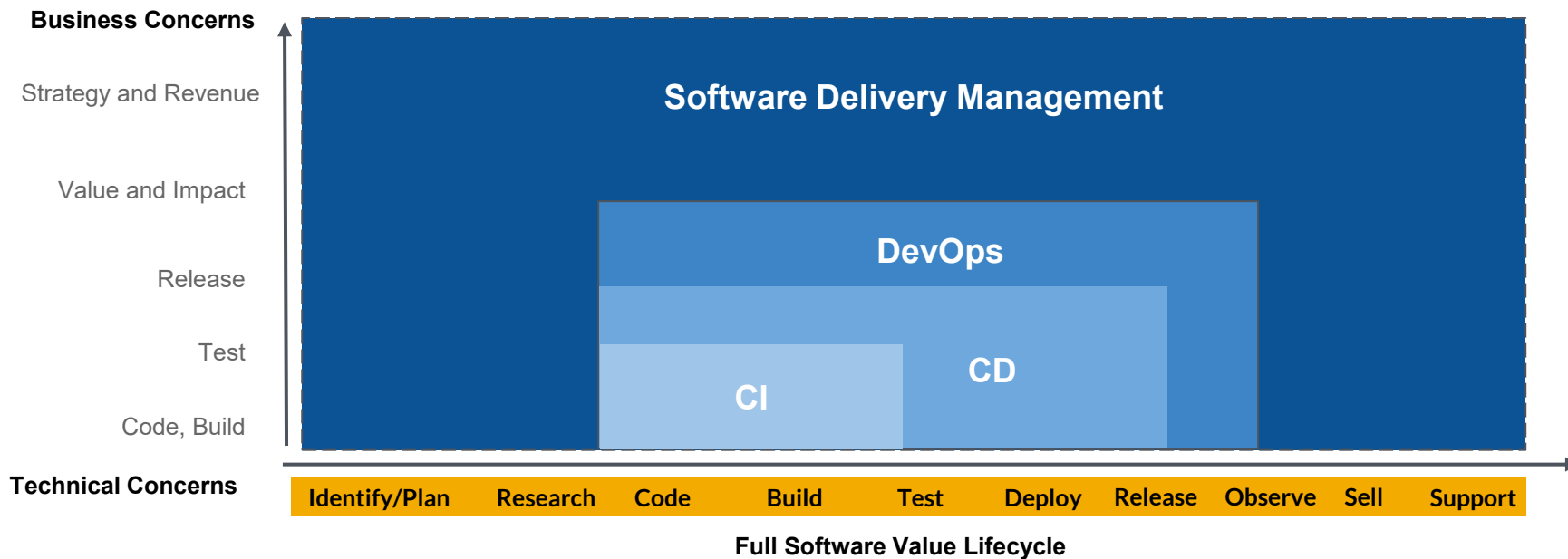
UNIVERSAL INSIGHTS

COMMON DATA



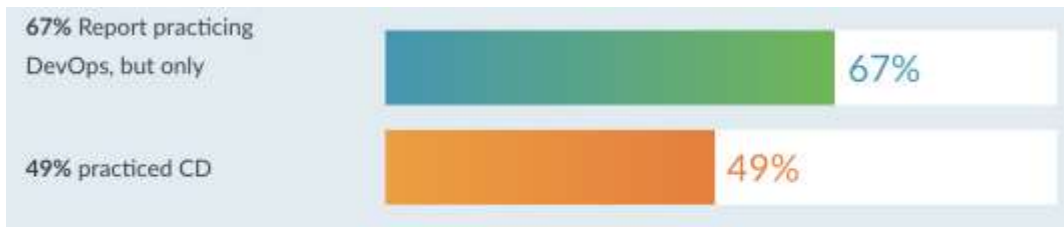
Existing Enterprise Tools

We have to Start with CI/CD and DevOps



But the reality is...

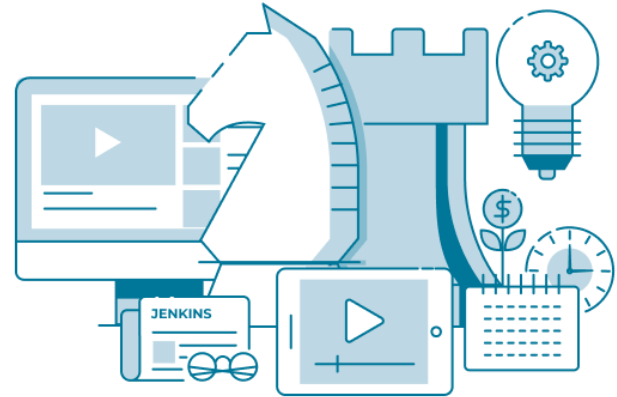
Far fewer people are truly practicing CI/CD than is widely reported!



The Path to the Future

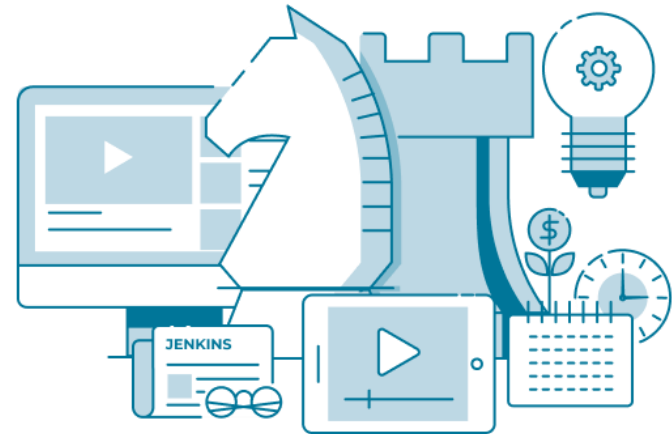
As the industry positions itself to build on DevOps practices with a Software Delivery Management strategy...

...it's more important than ever that we implement CI/CD best practices, and prepare for the future.



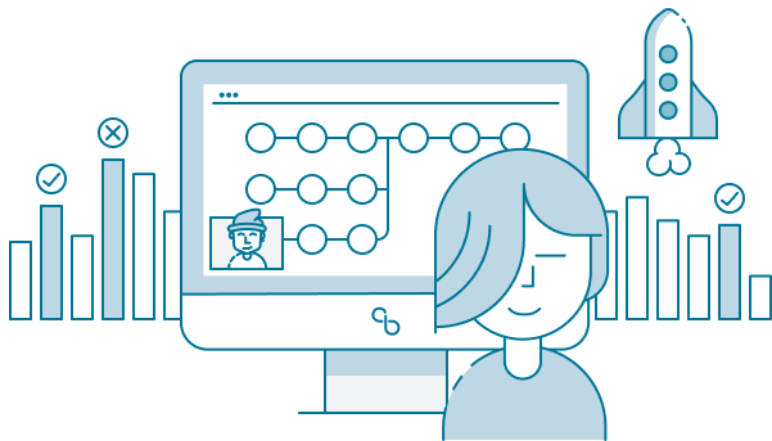
10 CI/CD Best Practices Summary

1. Track work items
2. Use source code management
3. Tags, not Branches
4. Automate the builds
5. Stop the line when the build breaks
6. **Validate and test**
7. Deploy
8. Improve incrementally
9. Collaboration
10. **Create a true DevOps culture**



Best Practice #1:
Track Work Items

Track Work Items



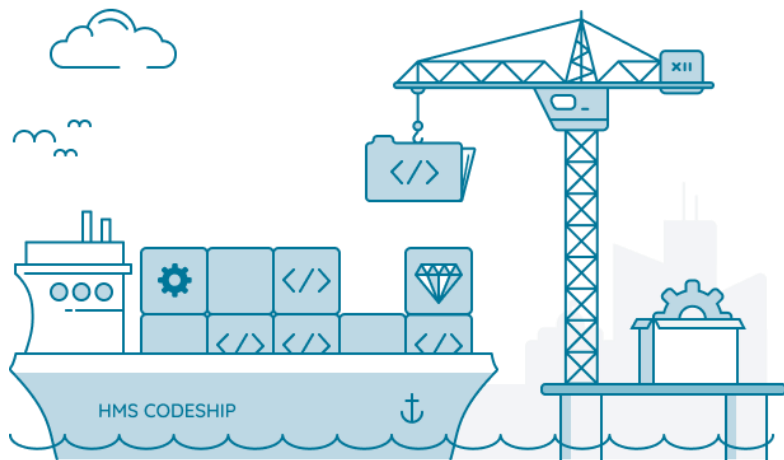
Use a system that lets you track everything related to your release:

- User Stories
- Bugs
- Infrastructure and environment

This is useful for CI and CD, but is absolutely critical as you move to DevOps and eventually Software Delivery Management (SDM).

Best Practice #2:
Use Source Code Management

Use Source Code Management (SCM)

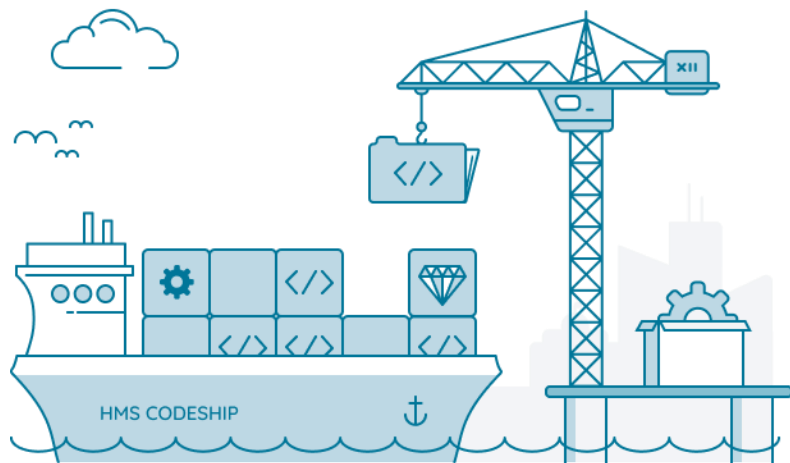


Use an SCM tool to track all the changes made for a release.

A decentralized version control system such as git
A centralized version control system such as SVN

Must consider support for new and legacy systems

Use Source Code Management

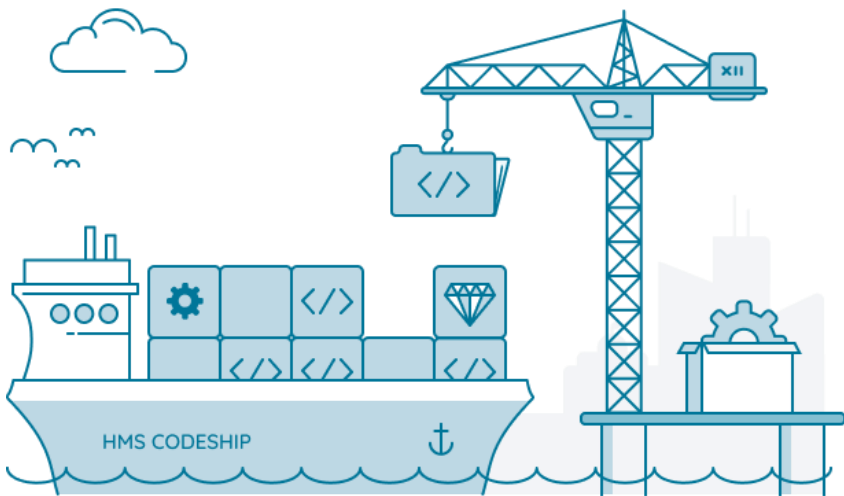


Encourage commit best practices

If culture is to not commit frequently, it won't matter. If a developer waits three weeks to commit or branches off for three weeks, they have delayed the integration and broken the principles

- Commit frequently
- Minimize duration of branches

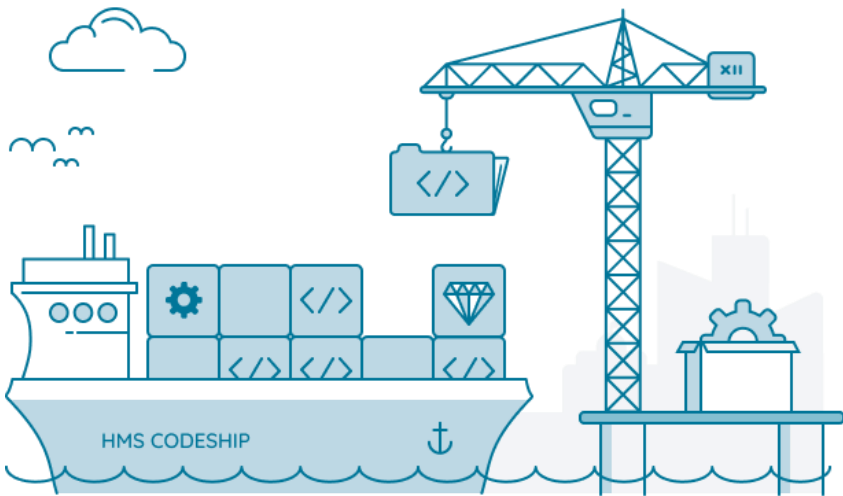
Use Source Code Management



Consider how triggers that launch builds are initiated.

- Trigger on every commit
- Trigger on developer branch
- Trigger on pull request
- Trigger on merge

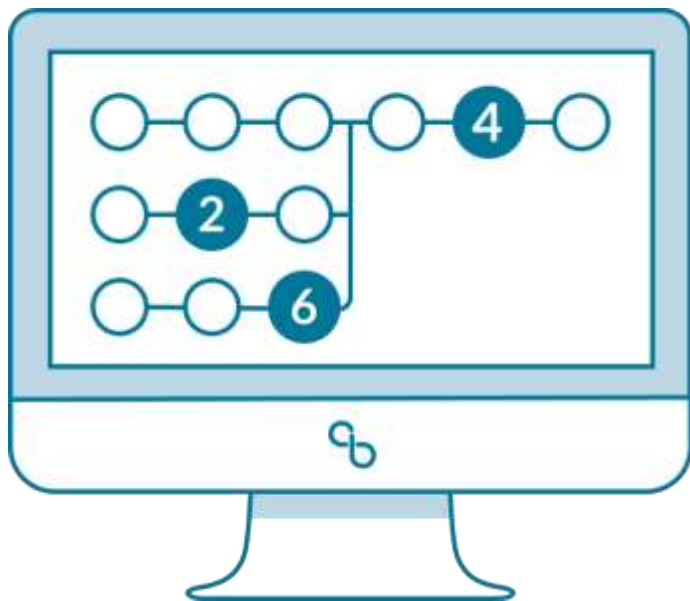
Best Practices -Source Code Management



- Use a descriptive commit message
- Link Commits to Issues (Feature, Bug, Task)
- Make each commit a logical unit
- Incorporate others' changes frequently
- Share your changes frequently
- Don't commit generated files

Best Practice #3:
Tags, not Branches

Tags, not Branches



Commit or Merge to the trunk frequently, or even better, always commit to the trunk/master

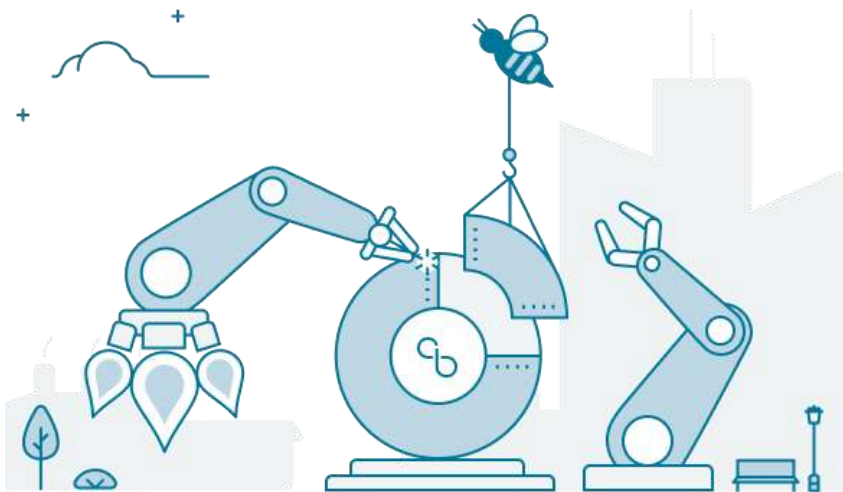
Developers often branch and maintain changes off of a trunk to manage releases.

But branching creates complexity that prevents everyone from working with a single source of truth, and introduces maintenance overhead

- Use tags to manage releases
- Implement functionality incrementally
- Use feature flags

Best Practice #4:
Automate the Builds

Automate the Builds



Centralize and automate the entire build process

- Compiling the source code
- Packaging the compiled code
- Container images

The build process should run for every commit.

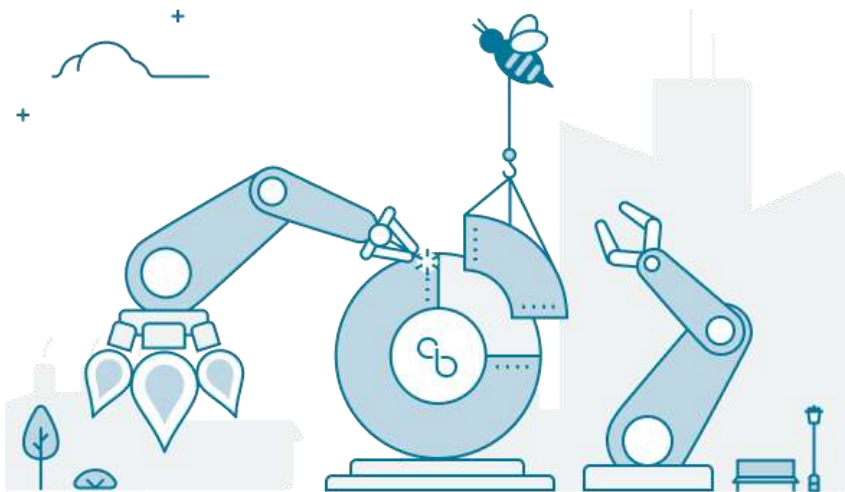
Automate the Builds

The build should run as quickly as possible

The builds should take no more than 5-10 minutes. Commit should be blocked during build, to support “stop-the-line” culture. If builds are long developers defer commits.

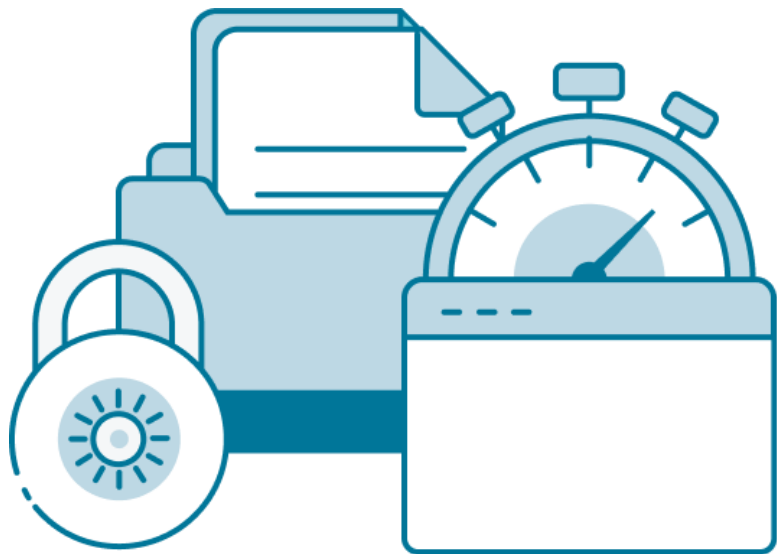
- Build
- Code scans, unit test
- Functional tests if possible

Promote the build to extended CI/CD loop for longer activities



Best Practice #6:
Validate and Test

Validate and Test



Organizations that don't validate every build are not practicing CI.

Organizations that do not continuously test are not practicing CD.

- Manage tests like code or as code
- Fail if the build if test thresholds aren't met
- Duplicate the production environment as much as possible

Best Practice #10:
Create a True DevOps Culture

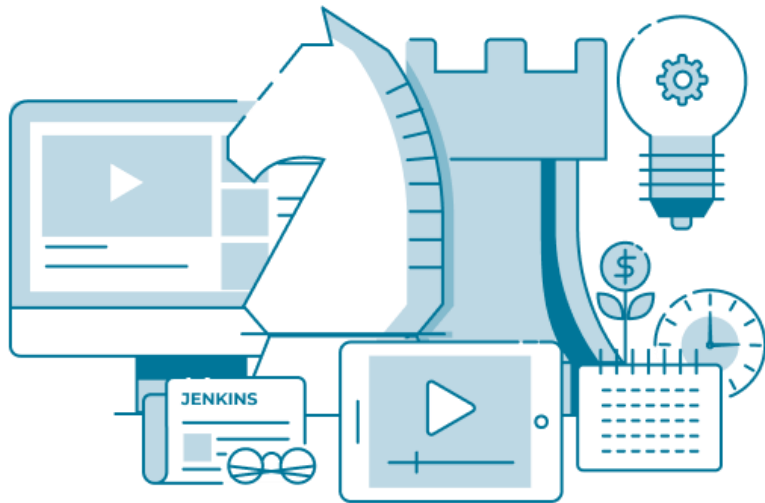
Create a True DevOps Culture

This may be the most difficult practice to implement, but no DevOps transformation is possible without it.

Starting a successful DevOps journey requires significant culture change, including:

- Executive support
- Training
- Funding
- A new mindset for everyone: the CIO/CTO, release managers, engineering managers, the ops team, and the dev team.

But..start, learn, and improve

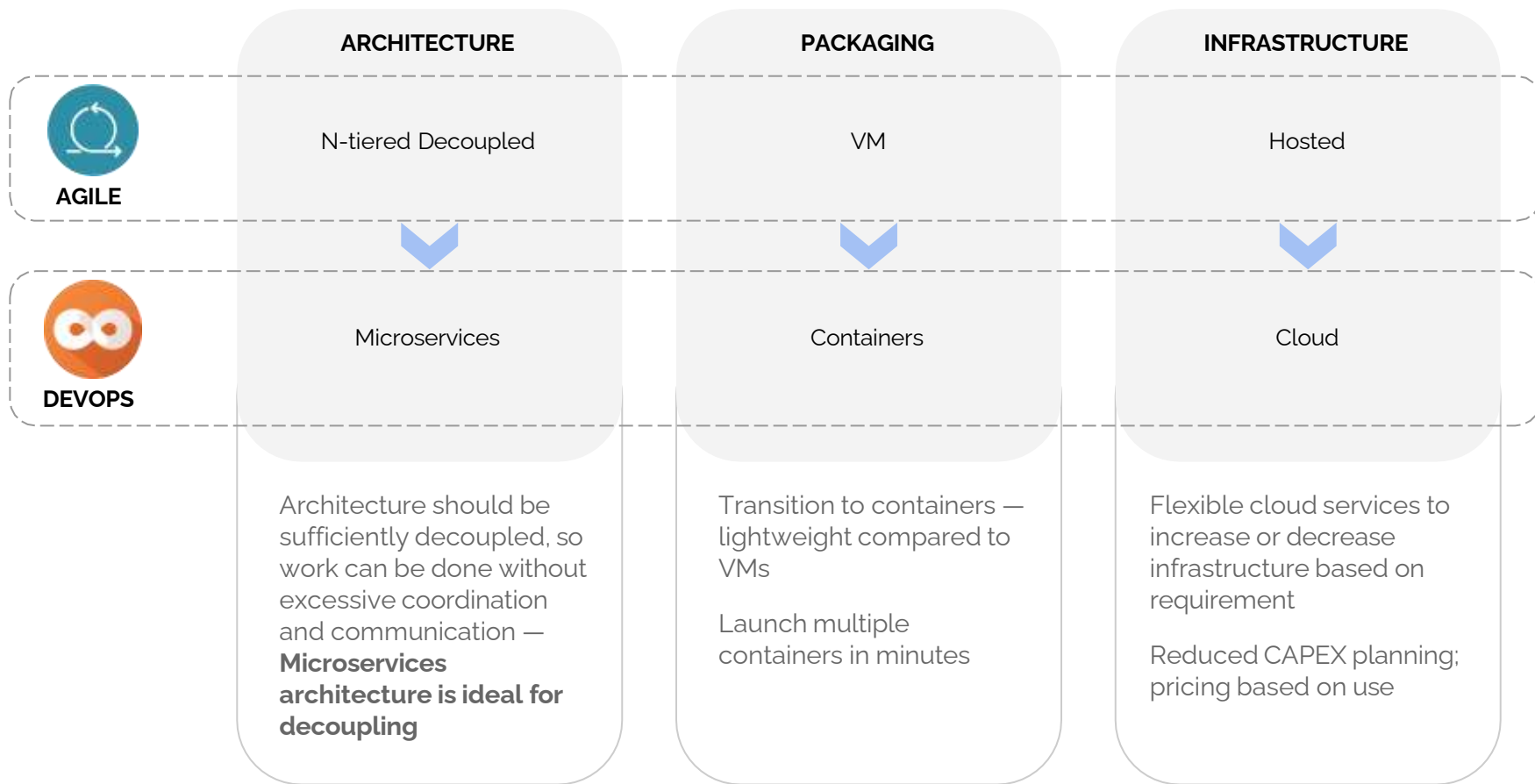


Thank You

Q & A

TECHNOLOGY TRANSFORMATION

SOFTWARE ARCHITECTURE - MOVING FROM REACTIVE TO SCALABLE



VERSION CONTROL

WHY?

- Standardize coding practices
- Keep a log and view changes
- Enables team to carry out development in parallel
- Faster debugging of deployment failures and production issues
- Infrastructure as Code: Anyone can create an environment using information from source control

WHAT?

- Code as well as infrastructure should be version controlled
- Operations-related artifacts
- Source code, automation scripts, deployment scripts, installation environment definition, infrastructure configuration and documents

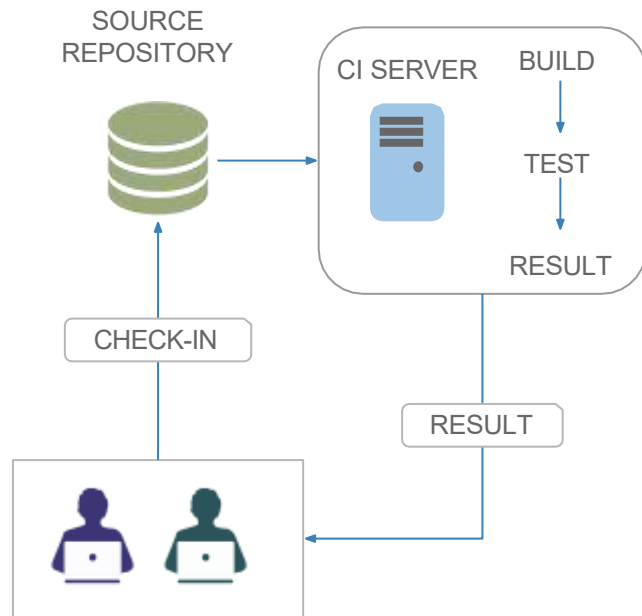
HOW?

- Version controlling tools
 - View difference between versions
 - Single source of truth

CONTINUOUS INTEGRATION

Integrating becomes exponentially difficult with an increase in branches or the number of changes in each branch.

- Merging into trunk should be part of everyone's daily work
- Create a comprehensive automated test suite: Automated tests to be written for new features, enhancements, and bug fixes
- Integrate in smaller batches
- Run locally before committing to CI server



AUTOMATED TESTING - BEST PRACTICES

- ✓ Code checked into version control must be automatically built and tested in a production-like environment
- ✓ Unit tests, acceptance tests, integration tests can be automated
- ✓ Unit and acceptance tests should run quickly — running them in parallel is recommended
- ✓ Automated testing on developer workstation can provide faster feedback
- ✓ Non-functional tests such as performance testing and security testing should be automated
- ✓ Automated tests should be reliable — false positives and unreliable tests create more problems than they solve
- ✓ A small number of reliable tests is better than a large number of unreliable tests

Start by building a small suite of reliable automated tests and expand coverage over time.

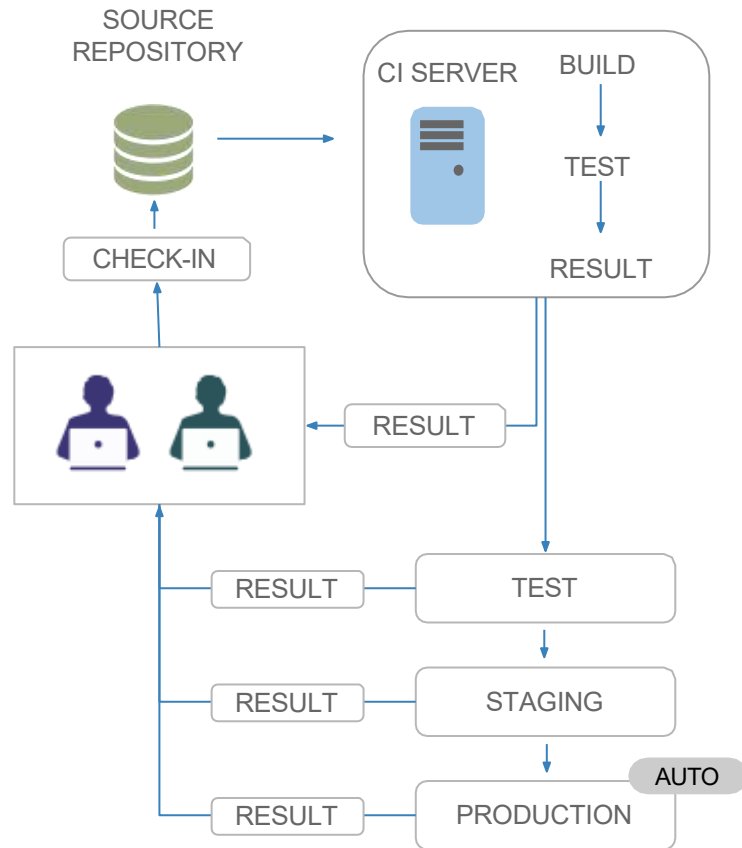
DEPLOYMENT PIPELINE

- Utilize visualization tools to monitor pipeline and ensure green-build state
- Create a virtual Andon Cord
 - If a change that causes build to fail is introduced, no new work should be accepted until the problem is fixed
 - Notify testers and developers when a problem needs to be fixed
- Static code analysis, duplication/test coverage analysis, and checking style
- Ensure staging environment is identical to production environment
- Some tools can be run from IDE or during pre-commit (via pre-commit hooks) to enable faster feedback
- Create containers as part of the build process

CONTINUOUS DEPLOYMENT

Developers should be able to deploy on-demand, enable multiple deploys per day.

- Keep code in a deployable state at all times
- Seamless feedback loop between users and developers
- Smaller sprints ensure faster turnaround time for bug fixes
- Select and share the right tools and procedures between teams
- Identify bottlenecks in deployment process and streamline over time



DEPLOYMENT STRATEGIES: INFRASTRUCTURE-BASED

Blue-Green Deployment

- Deploy complete application components, services twice
- Old version in blue and new version in green side by side
- To cut over to the new version and roll back to old, change the load balancer or router setting

Canary Release

- Incrementally upgrade to the latest version
- First build can go to employees, then to a larger group, and finally to everyone
- If a problem is discovered in an early stage, build goes no further

Cluster Immune System

- Monitors critical system metrics when a new version is rolled out in a canary release.
- Automatically rolls back the deployment in case of high stats

DEPLOYMENT STRATEGIES: APPLICATION-BASED

Feature Toggles

- Control who can access a new feature to test a feature on production with a select group of users.
- Implement a toggle router to dynamically control which code path is live
- Toggle Router can make decisions based on environment-specific configurations
- Helps release near bug-free features

Dark Launches

- Releasing production-ready features to a subset of users first
- Helps get real user feedback, test for bugs, and assess infrastructure performance
- In case something goes wrong during deployment, it's easy to roll back the changes and fix the problem with the old infrastructure/code still in place,

ENVIRONMENT: DOS & DON'TS



Developers should have the ability to **create production-like environments on-demand.**

Automate environment creation process. This applies to development, testing, and production environments.

Utilize tools such as chef to **automatically configure newly provisioned environments.**

Make infrastructure easier to build than repair.

Entire application stack and environment can be bundled into containers. **Package applications into deployable containers.**

Verify that the application runs as expected in a production-like environment before the end of a sprint.



Inconsistently constructed environments and not putting changes back to version control can create problems.

Immutable infrastructure - manual changes to production environment are not allowed.

MONITORING

- Have an integrated monitoring system for Dev and Ops
- Move all monitored data to a central location
- Derive metrics from monitored data
 - Plot metrics as graphs
 - Display deployment events on the same graph to correlate problems with deployments
 - Statistically analyze collected metrics to identify deviations.
Example: Generate an alert if the metric (e.g. page load time) is 3 standard-deviations away from the mean (this assumes the metric has a Gaussian distribution)
 - ✓ Another strategy is outlier identification
Example: In a cluster of thousands of nodes, if the performance metrics of a node deviates from the normal, it can be taken down
- Alerts: Generate alerts only for indicators that predict outages. Too many alerts could cause alert fatigue.
- Anomaly detection: Use statistical tools (Excel, SPSS, SAS, R etc.) on datasets to find anomalies. *Example:* If orders fall below 50% of the normal (expected number of orders based on historical trend) on Wednesday morning
- Kolmogorov-Smirnov test: Find similarities/differences in seasonal/periodic data

What should be monitored?

End-to-end monitoring of the entire software stack is essential.

- Applications
- Databases
- Servers and networks
- Builds
- Automated tests
- Deployments
- All environments (development, testing, staging, production)

Tools

Graphite | Grafana | Nagios

Devops Testing

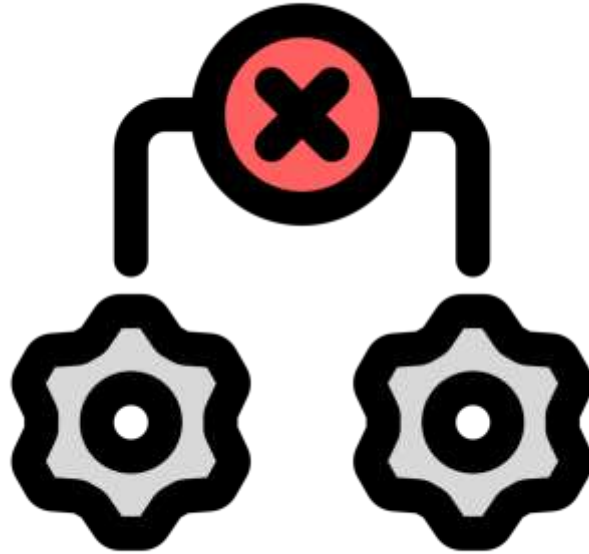
- Process of improving software quality using continuous automated tests.
- DevOps testing tools and methods help software delivery teams deliver changes faster, without affecting safety.
- Different types of tests combine throughout the delivery lifecycle to ensure complete coverage
- **Key Component: Automated Testing.**
- **Key Component: Continuous Automated Testing.**



Devops Testing vs Traditional Test Frameworks

- **Automation:** Instead of manual testing, all tests are fully automated, typically as part of a CI/CD pipeline. This improves speed, consistency, and safety — changes with failing tests are prevented from being deployed.
- **Continuous by design:** Tests run continually throughout the DevOps lifecycle to deliver actionable results earlier. This contrasts with traditional approaches, where tests may only run at specific points in the software development lifecycle process, such as after all the changes for a feature have been completed.
- **Visibility:** Test results, failures, and issues are clearly visible to all stakeholders. This improves workflow efficiency while making it easier to assess software quality accurately.
- **Collaboration:** DevOps testing advocates close collaboration between developers and operations teams to produce more relevant test suites. For instance, developers may create new test cases based on operational requirements, whereas operators might run IaC tests to ensure infrastructure is provisioned in the correct state for deployment.
- **Comprehensiveness:** Traditional software testing usually focuses on unit, integration, and E2E tests for source code. DevOps testing aims to verify correct operations throughout the entire software delivery lifecycle, including infrastructure, security, and compliance tasks.

Common Challenges



Environment inconsistencies between development, testing, and production can lead to flaky test results or missed bugs.

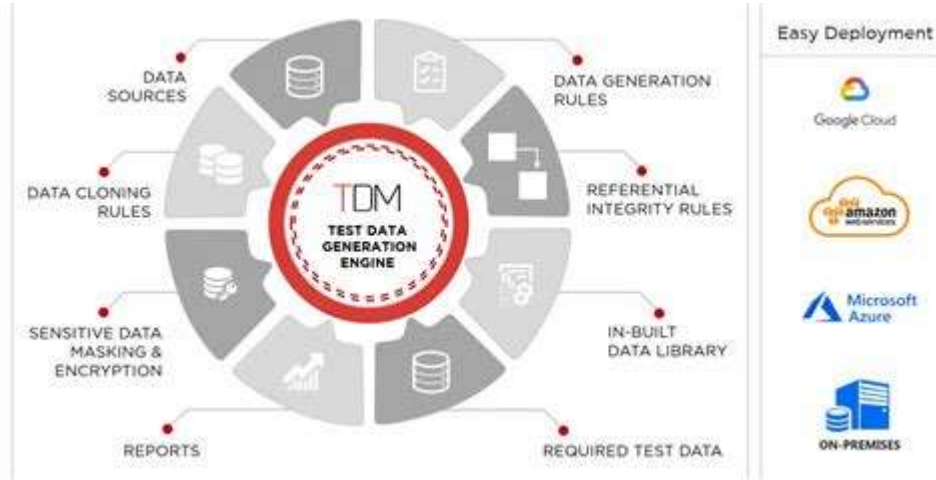
Common Challenges

Typical Enterprise DevOps Toolchain



Toolchain integration issues can slow pipelines or break automation if CI/CD tools, test runners, and environments aren't well aligned.

Common Challenges



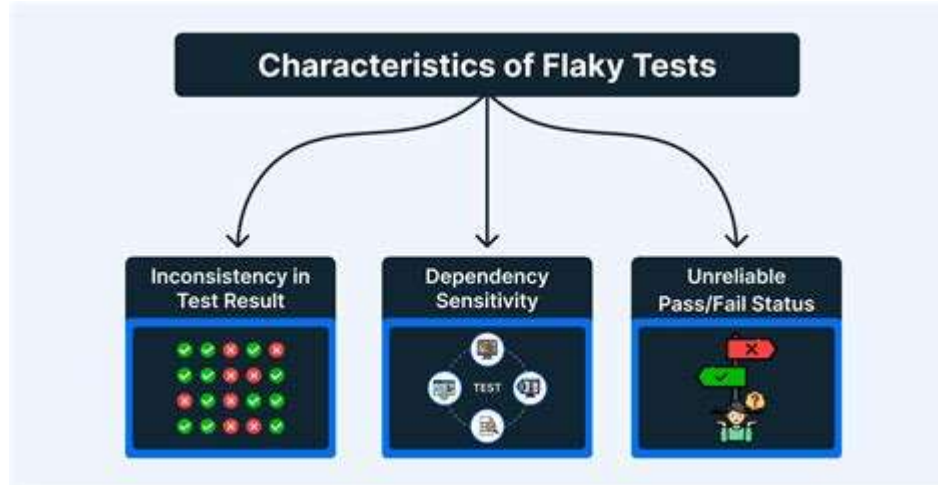
Test data management, especially for large-scale or secure systems, often creates bottlenecks or privacy concerns.

Common Challenges



Maintaining test coverage in fast-moving CI/CD cycles is difficult, especially with frequent changes.

Common Challenges



Flaky tests from timing issues or parallelism reduce trust in test results and delay releases.

Test Types

Test type	Primary purpose	Scope / Focus	Typical environment	Key benefits	Example use case
Unit test	Validate the correctness of individual functions or methods	Single, isolated unit of code	Local dev or CI job	Fast feedback, easy maintenance, catches logic errors early	Confirm a sort() function returns values in ascending order Service A writes a record; Service B reads and transforms it correctly
Integration test	Verify interactions between integrated components	Interfaces between two or more modules/services	Dedicated integration env or CI pipeline	Detects contract mismatches and data-flow issues	Login endpoint returns token and UI redirects on valid credentials
Functional test	Ensure a feature meets business requirements at API/UI level	End-user features & core business logic	System/staging env with near-real data	Confirms feature completeness from user perspective	

Test Types

Test type	Primary purpose	Scope / Focus	Typical environment	Key benefits	Example use case
End-to-End (E2E)	Validate full user workflows across the entire stack	Complete application + external services	High-fidelity staging or prod-like CI env	Provides confidence that critical journeys work as expected	New user signs up, adds items to cart, and checks out via payment gateway
Acceptance test	Demonstrate that deliverables satisfy stakeholder expectations (often BDD) Guard against re-introduction of previously fixed defects	High-level requirements and workflows expressed in business language	UAT / pre-production; often automated in CI	Shared "definition of done," enables stakeholder sign-off	BDD scenario for discount rule passes and product manager approves release
Regression test		Areas historically affected by bugs	CI/CD pipeline test stage	Maintains stability across releases	After refactor, suite confirms checkout still processes payments correctly

Test Types

Test type	Primary purpose	Scope / Focus	Typical environment	Key benefits	Example use case
Fuzz test	Expose unhandled errors by sending random or malformed input	APIs, parsers, input-handling components	Local or CI fuzzing harness; sometimes staging	Reveals edge-case crashes and security flaws	Fuzz a REST endpoint with random JSON and verify no 5xx or crashes
Security test	Detect vulnerabilities and misconfigurations across code and runtime	Code (SAST), running apps (DAST), deps (SCA), containers, IaC	Throughout SDLC (CI/CD, runtime)	Ensures security hygiene and compliance, reduces risk	SAST flags SQL injection; container scan reports outdated OpenSSL

Test Types

Each of these test types can be found within multiple workflows in the DevOps lifecycle. You'll typically need tests for each of the following areas to attain full coverage:

- **Code tests** – Use unit, integration, functional, and E2E tests to verify your code works as expected.
- **API tests** – Test API endpoints to ensure correct results are returned. You may also use fuzz tests to check how APIs handle improper input.
- **Interface tests** – Functional and E2E tests can help you check that user interfaces render correctly and trigger appropriate actions when controls are used.
- **IaC tests** – Writing tests for your IaC config files lets you avoid infrastructure misconfigurations.
- **Policy tests** – Writing tests for policy-as-code rules helps ensure security and compliance rules are correctly enforced. Consider using fuzz testing to check how your policies behave with unusual inputs.

Implementing a DevOps Testing Strategy

Include multiple test types

Easy Developer Access to Test Results

Continual iterative improvement

Implementing DevOps testing strategy



Establish a DevOps
testing culture



Assess
which workflows
need testing



Select your
test tools



Plan test cases
during the DevOps
requirements stage



Implement
continuous testing

Lets have fun with Tools

<https://zebrunnertest.com/blog-posts/140-tools-for-software-testers-choose-the-most-suitable-for-your-team>

Lets have fun with Tools

Icon Style
 SonarQube
 PMD
 Checkstyle
 ESLint
 Bandit
 Brakeman
 Semgrep
 Flawfinder
 cppcheck
 GolangCI-Lint
 Fortify
 Veracode
 Checkmarx
 Coverity
 CodeSonar
 AppScan
 Klocwork
 Snyc
 DeepSource
 CodeQL
 SonarCloud

💡 Description / Focus Area
Code quality + security across many languages
Java rule-based static analysis
Java style and convention enforcement
JavaScript/TypeScript linting
Python security analyzer
Ruby on Rails security scanner
Lightweight, customizable multi-language scanner
C/C++ vulnerability scanner
C/C++ static analysis
Aggregates Go linters
Enterprise-grade SAST with CI/CD integration
Cloud-based static analysis
Scalable enterprise scanning
Strong in C/C++; safety-critical systems
Advanced dataflow and binary analysis
IBM's SAST + DAST integration
Embedded systems and compliance
Developer-friendly; Git + IDE integration
Code health + security for Python, Go, Ruby
Semantic code analysis; GitHub-native
SonarQube's cloud offering

IMPORTANCE OF TESTING IN DEVOPS:

- Testing plays a critical role in DevOps by ensuring the quality, reliability, and security of software applications throughout the development lifecycle.
- By conducting comprehensive testing, organizations can identify and address defects early in the development process, minimizing the risk of costly errors and delays later on.

SHIFT-LEFT TESTING APPROACH

- The shift-left testing approach is a fundamental principle of DevOps that advocates for moving testing activities earlier in the development cycle.
- By shifting testing left, teams can detect and resolve issues sooner, reducing the time and effort required for debugging and rework.



INTEGRATION OF TESTING INTO THE DEVOPS PIPELINE:

- In DevOps, testing is seamlessly integrated into the development pipeline, with automated tests running at various stages of the process.
- Testing is incorporated into continuous integration (CI) and continuous delivery (CD) pipelines, ensuring that code changes are thoroughly tested before deployment.
- By integrating testing into the DevOps pipeline, organizations can achieve faster feedback loops, accelerate delivery cycles,

Testing Strategies in DevOps

- Test-Driven Development (TDD)
- Behavior-Driven Development (BDD)
- Continuous Testing
- Canary Testing:
- Blue-Green Deployment



Test-Driven Development (TDD):

Behavior-Driven Development (BDD) is an extension of TDD that focuses on the behavior of the system from the user's perspective. BDD involves defining tests in plain language that describe the expected behavior of the application. This encourages collaboration between stakeholders and helps ensure that development efforts are aligned with business requirements.

Behavior-Driven Development (BDD):

Test-Driven Development (TDD) is a software development approach where tests are written before the code. Developers write failing tests based on requirements, then write code to pass those tests. This approach promotes a focus on writing clean, modular, and testable code.



A close-up photograph of a person's hand, wearing a dark pinstriped suit jacket and a blue tie, tipping a red wooden domino. The red domino is the first of a line of seven wooden dominoes standing upright on a dark surface. The background is a dark teal color with a subtle pattern of small white dots.

CONTINUOUS TESTING

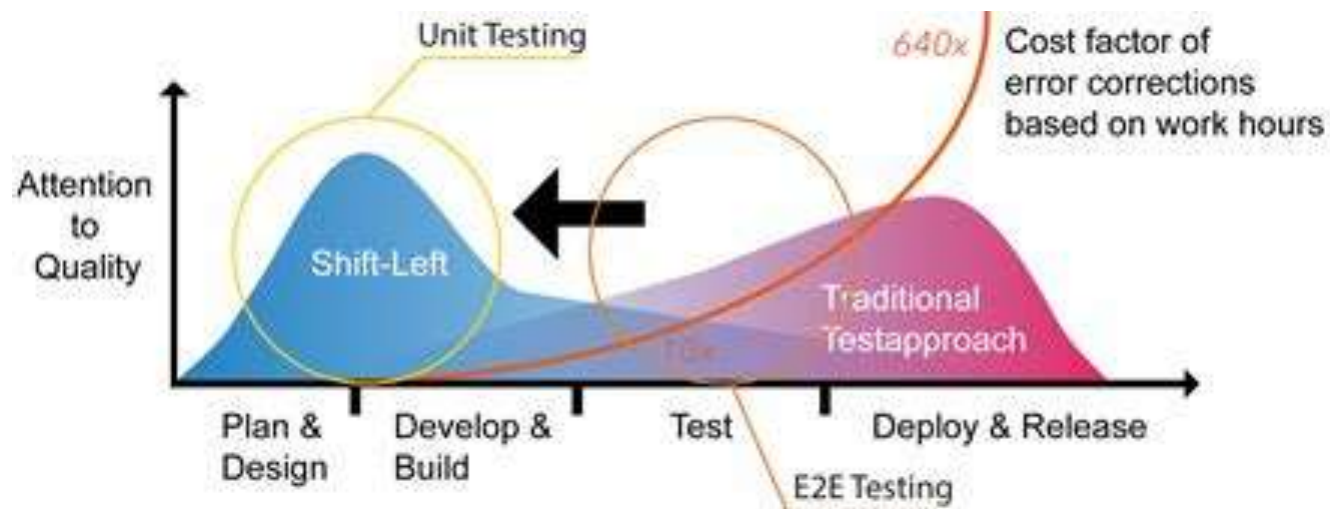
Continuous Testing is the practice of executing automated tests throughout the software delivery pipeline. Tests are run automatically at various stages, including unit tests, integration tests, and end-to-end tests. Continuous Testing helps identify defects early, maintain code quality, and validate changes before deployment

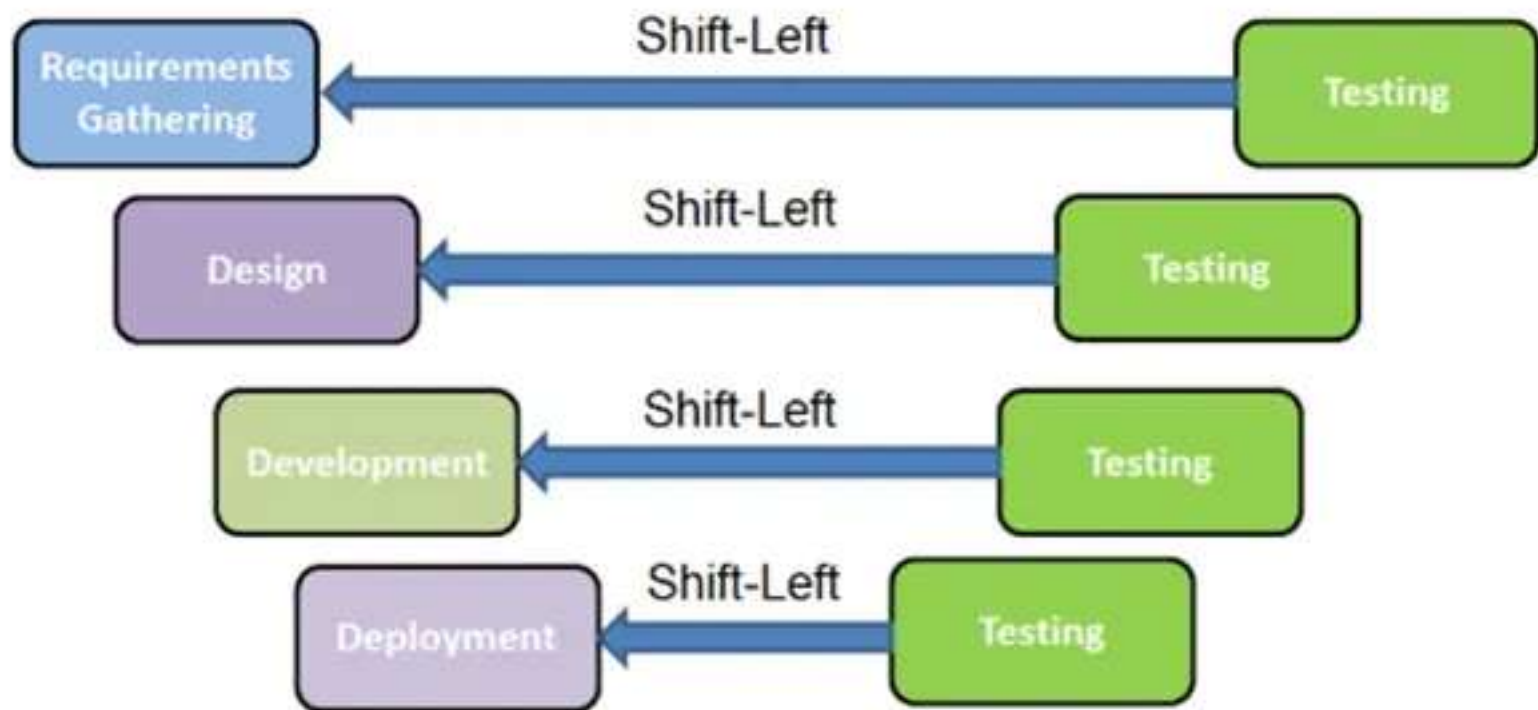
CANARY TESTING

Canary Testing is a deployment strategy where changes are gradually rolled out to a small subset of users or servers before being deployed to the entire infrastructure. By monitoring the performance and stability of the canary release, teams can assess the impact of changes and mitigate risks before a full deployment.

BLUE-GREEN DEPLOYMENT

Blue-Green Deployment is a deployment technique where two identical production environments, "blue" and "green," are maintained. Only one environment is active at a time, while the other remains idle. Changes are deployed to the inactive environment, allowing for zero-downtime releases. Once the new version is validated, traffic is switched to the updated environment. This approach reduces deployment risk and enables fast rollback in case of issues.





The background of the slide features a grayscale photograph of a city skyline with several tall skyscrapers. A large, thick, red brushstroke is painted across the right side of the image, partially obscuring the buildings. The title text is written in a white, cursive script font and is positioned over the left side of the red brushstroke.

Comprehensive Guide to Java Testing Frameworks

John Doe

Software Engineer

Introduction to Java Testing Frameworks

Unit Testing

Unit testing ensures individual code components function properly before integration. It significantly reduces bugs and facilitates easier maintenance by identifying issues early in the development process.

Integration Testing

Integration testing evaluates the interaction between multiple components to uncover interface issues. This step is crucial for ensuring that combined modules work together effectively and meet system requirements.

Performance Testing

Performance testing assesses the responsiveness and stability of software under varying loads. Ensuring an application can handle high traffic and uses system resources efficiently is vital for user satisfaction.

Importance of Testing in Software Development

Reduces Bugs

Effective testing minimizes defects and improves software quality.

Enhances Performance

Testing ensures the application performs well under all conditions.

Builds User Confidence

Thorough testing instills trust and reliability in end-users.

Facilitates Team Collaboration

Testing encourages communication among developers, testers, and stakeholders.

Promotes Code Reusability

Unit tests promote refactoring and enhance code maintenance efforts.

Saves Cost

Identifying issues early reduces expenses associated with fixing them.

Boosts Productivity

Automated tests speed up the development and deployment processes.

Improves Compliance

Testing ensures adherence to industry regulations and standards.

Java Testing Framework Overview

Unit Testing

Unit testing frameworks like JUnit allow developers to test individual components for functionality and reliability.

Integration Testing

Integration testing frameworks help ensure that different components of the Java application work together as intended.

Test Automation

Java testing frameworks facilitate automated testing processes, allowing faster feedback and increased test coverage.

Test Reporting

Frameworks provide valuable test results and logs, making it easier to track down issues and improvements over time.

JUnit: Core Features and Benefits

Annotations

Utilize `@Test`, `@Before`, and `@After` for streamlined testing workflows.

Assertions

Implement various assertion methods for verifying expected results effectively.

Test Suites

Organize multiple test cases into suites for efficient execution and management.

Parameterized Tests

Run the same test with different input values for extensive coverage.

Integration

Seamlessly integrate with build tools like Maven and Gradle for automation.

TestNG: Key Features and Advantages

Data Providers

Facilitates parameterized tests by providing data from various sources.

Parallel Execution

Enhances efficiency by allowing concurrent test execution across multiple threads.

Test Annotations

Simplifies test configuration through clear and structured annotations.

Integrated Reporting

Automatically generates detailed reports aiding in test analysis and debugging.



Comparison of JUnit and TestNG

Simplicity

01

JUnit is known for its straightforward approach to writing tests easily.

Annotations

02

JUnit provides essential annotations like `@Test`, `@Before`, and `@After`.

Timeouts

03

JUnit allows the specification of timeout values for test execution.

Run Order

04

JUnit does not guarantee the order of test execution unless specified.

JUnit

C
O
M
P
A
R
I
S
O
N

TestNG

Flexibility

01

TestNG is more flexible and allows for powerful features and configurations.

Annotations

02

TestNG offers a wider range of annotations compared to JUnit.

Parameterization

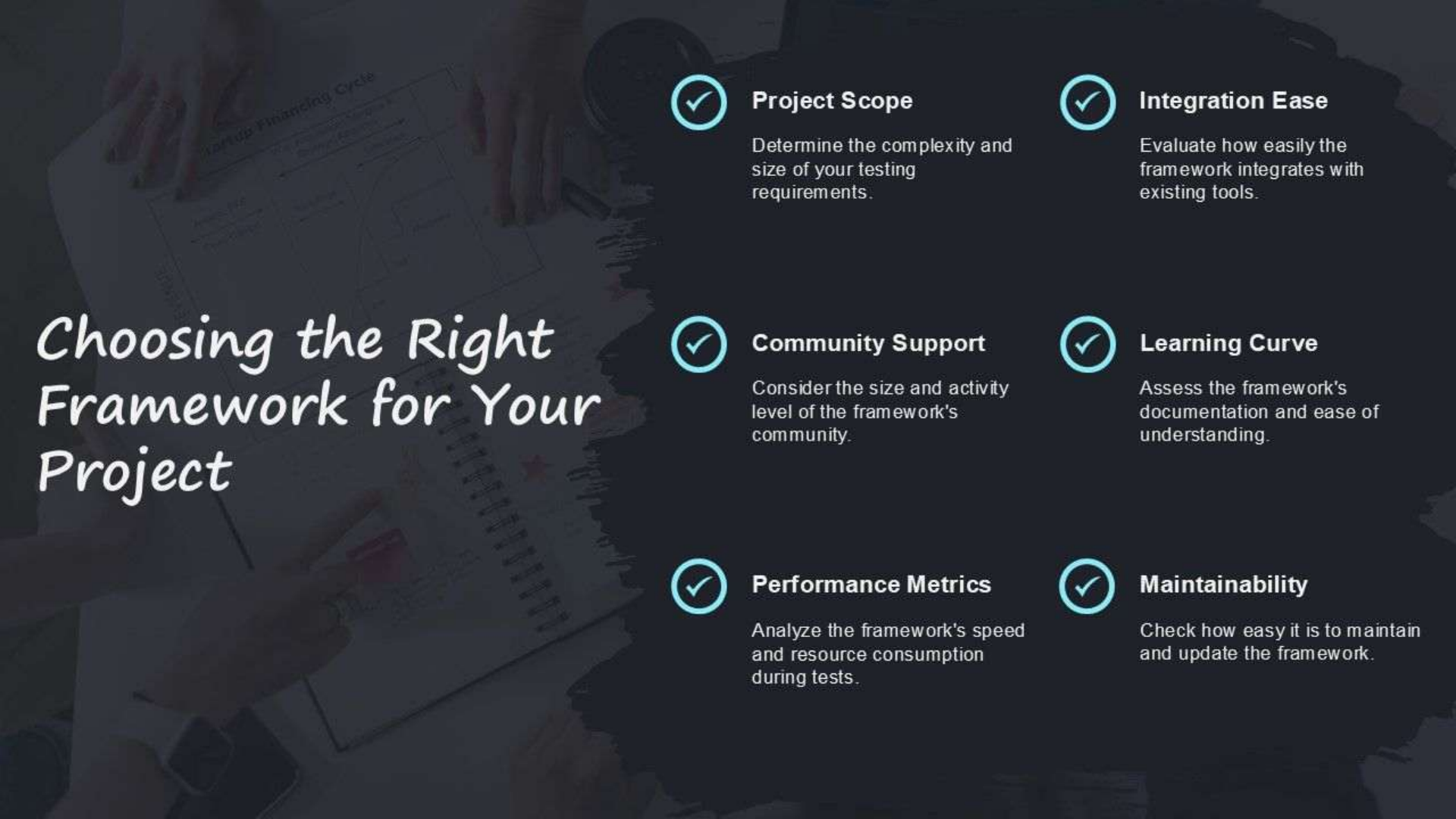
03

TestNG easily supports parameterized tests through XML configurations.

Parallel Runs

04

TestNG allows tests to be run in parallel, enhancing speed and efficiency.



Choosing the Right Framework for Your Project



Project Scope

Determine the complexity and size of your testing requirements.



Integration Ease

Evaluate how easily the framework integrates with existing tools.



Community Support

Consider the size and activity level of the framework's community.



Learning Curve

Assess the framework's documentation and ease of understanding.



Performance Metrics

Analyze the framework's speed and resource consumption during tests.

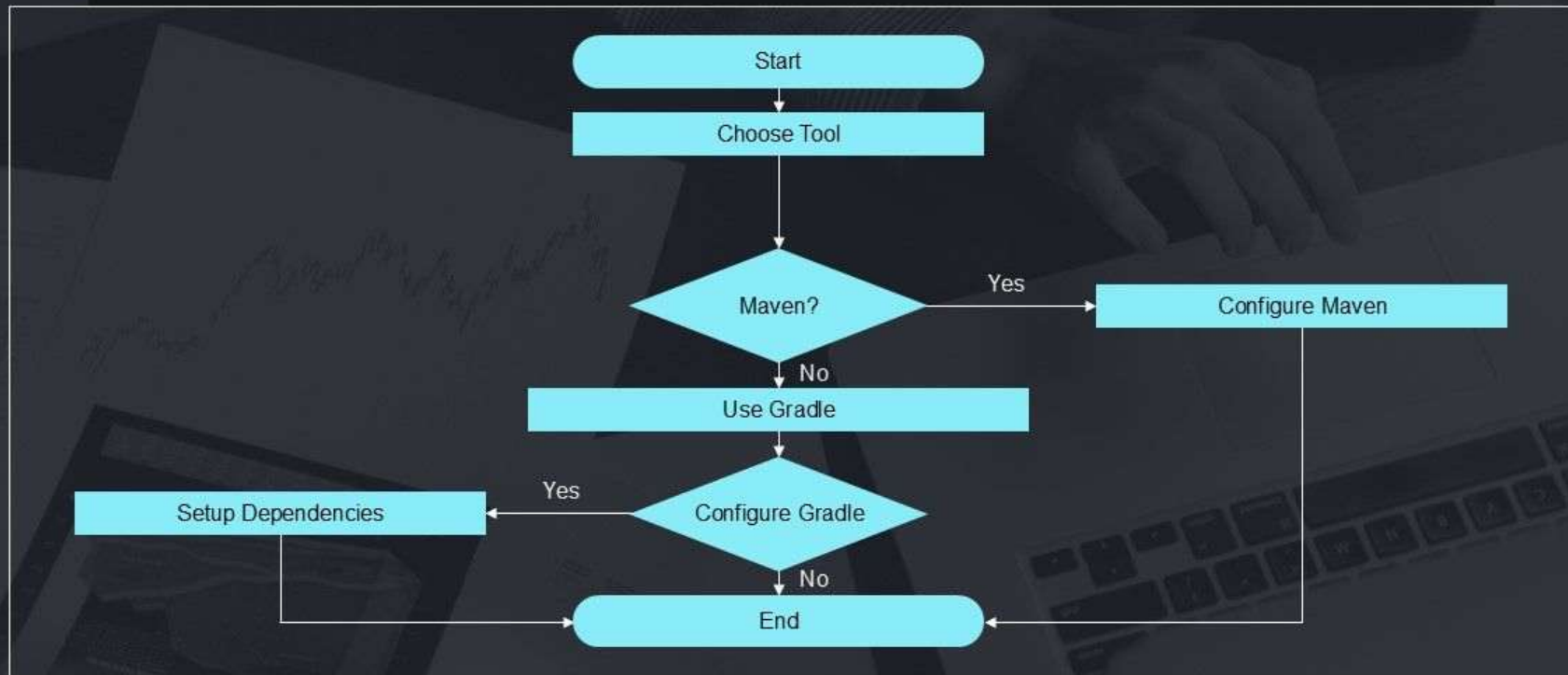


Maintainability

Check how easy it is to maintain and update the framework.

Integration with Build Tools: Maven and Gradle

Flow Chart



This is a sample flowchart for this slide. Please rearrange the flowchart to convey your message.

Writing Effective Unit Tests with JUnit



Test Naming

Use descriptive names that convey the purpose clearly.



Assertions

Make assertions precise to verify expected behaviors rigorously.



Isolation

Ensure tests run independently without external dependencies involved.



Setup Method

Utilize setup methods to prepare test conditions before execution.



Clear Failures

Provide clear feedback in case of test failures for troubleshooting.



Mocking

Employ mocking frameworks to simulate dependencies effectively.



Code Coverage

Aim for high code coverage to assess the thoroughness of tests.

Parameterization in JUnit and TestNG

Data Driven

Parameterization enables the execution of the same test method with multiple data sets, leading to comprehensive testing and reduced code duplication in test cases.

Efficiency Boost

Using parameterization allows for faster test execution times by minimizing redundancy and facilitating batch testing of multiple inputs simultaneously.

Improved Coverage

Parameterized tests can enhance test coverage by validating various edge cases and scenarios without the need for repetitive test method definitions.

Handling Exceptions in Tests

Use Try-Catch

Encapsulate test logic within try-catch to handle possible exceptions.

Assert Failures

Use assertions to capture failures and ensure tests are accurate.

Custom Exceptions

Implement custom exception classes for better clarity in handling errors.

Error Logging

Integrate logging to capture detailed error information during testing.

Parameterized Tests

Utilize parameterized tests to cover various exception scenarios.

Clean Up Resources

Ensure proper cleanup of resources in finally block after tests.

Test Case Design Techniques

Boundary Value

Focus on test cases at the edges of input ranges.

01

Equivalence Partitioning

Group inputs to reduce number of test cases effectively.

02

Decision Table

Use tables to represent complex logical scenarios clearly.

03

State Transition

Test the system's change in state based on events.

04



01

Mock Creation

Use `Mockito.mock()` to create a mock instance of a class.

02

Stubbing Methods

Utilize `Mockito.when()` to define behavior for mocked methods.

03

Verification

Verify interactions with mocks using `Mockito.verify()`.

04

Argument Captors

Capture method arguments using `ArgumentCaptor` for assertions.

Advanced Mocking with Mockito

Testing Asynchronous Code in Java

Mockito Usage

Utilize Mockito for mocking asynchronous calls in tests.

01

CompletableFuture

Test CompletableFuture using AssertJ to handle results.

02

CountDownLatch

Implement CountDownLatch to synchronize asynchronous actions.

03

Thread Pools

Use ExecutorService to manage threading in tests.

04

Awaitility Framework

Adopt Awaitility to write fluent asynchronous testing assertions.

05



Code Coverage Tools for Java Testing

Test Coverage

85%



Execution Time

30s



Bug Density

2%



Pass Rate

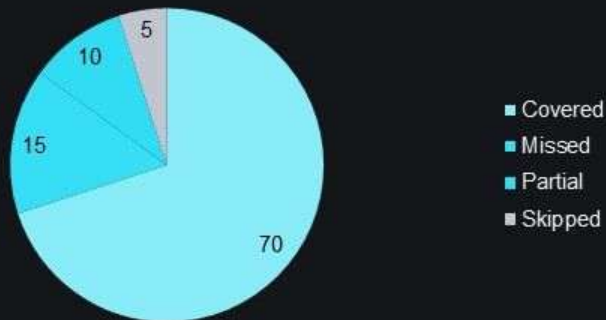
92%



Java Code Coverage



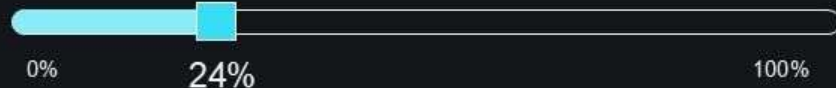
Coverage Breakdown



Executed Test Cases

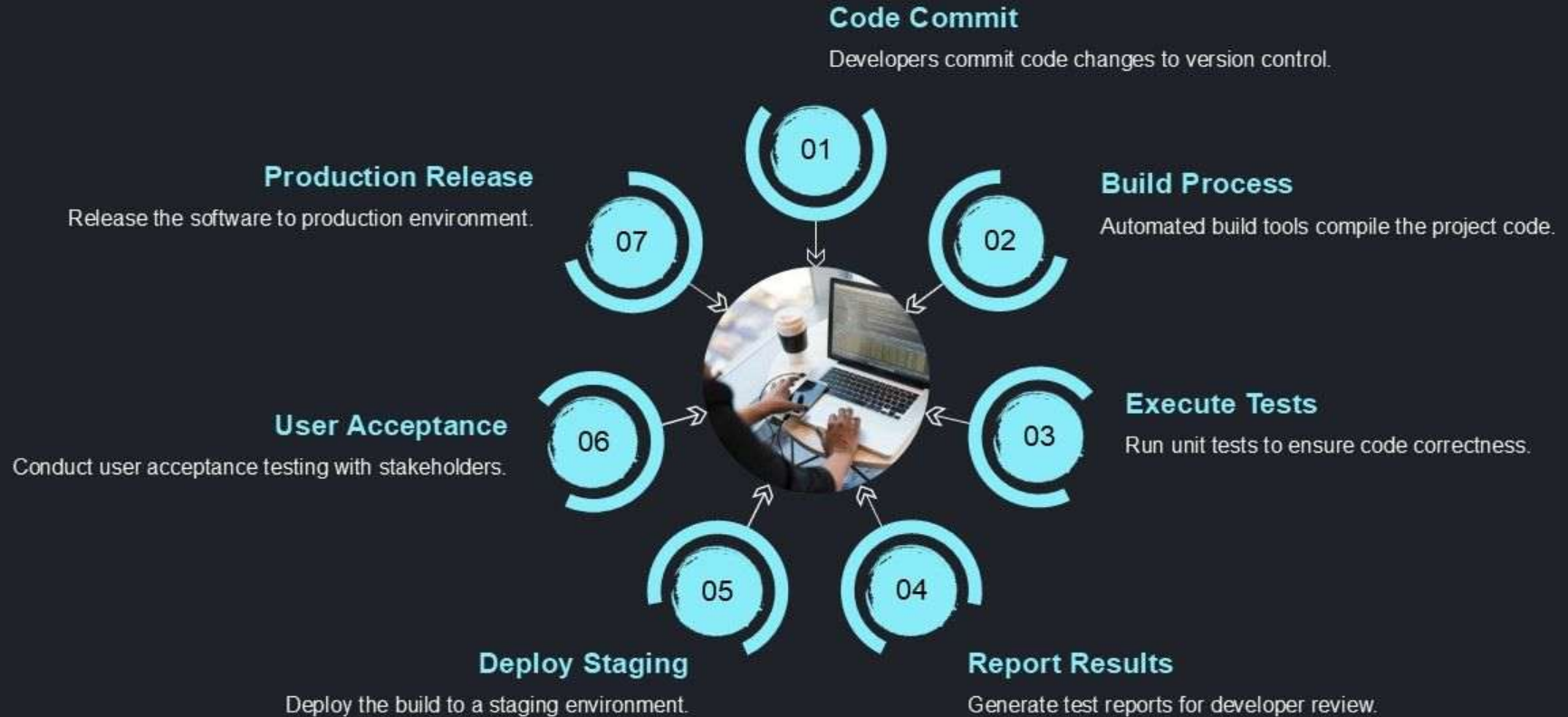


Resolved Bug Rates



This is a sample dashboard and is 100% editable. Please edit the metrics according to your message.

Continuous Integration and Testing



Behavior Driven Development (BDD) with JBehave

User Stories

JBehave allows teams to write clear user stories that specify the expected behavior of a system, fostering better communication among developers and stakeholders.

Automated Tests

By utilizing JBehave, developers can create automated tests from user stories, enabling faster feedback and ensuring that requirements are met throughout the development process.

Collaboration

JBehave promotes collaboration between technical and non-technical team members by using a shared language in behavior specifications, leading to enhanced project efficiency.

Real-World Case Studies of Java Testing



Problem Faced

Inconsistent results from unit testing processes.



Solution Offered

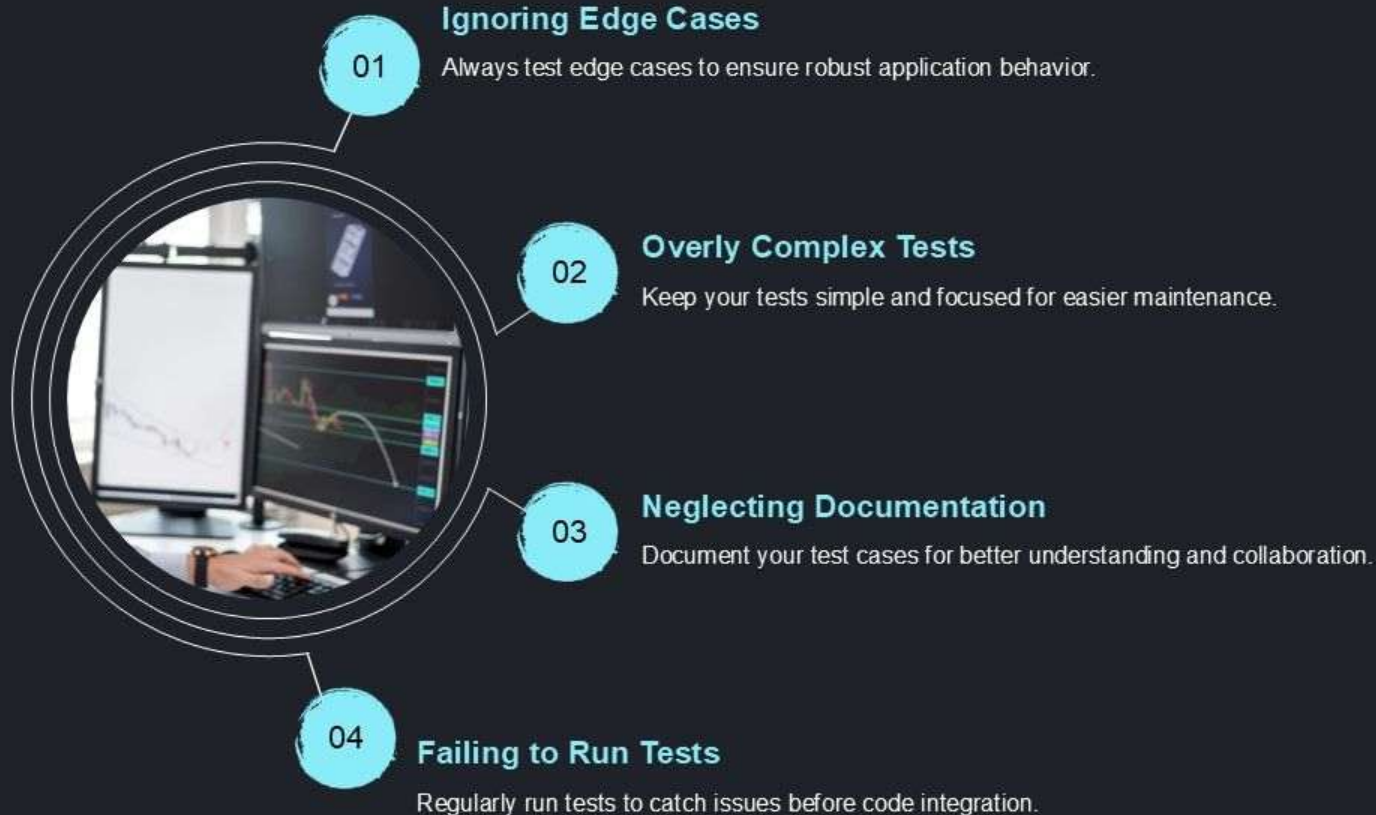
Implemented the JUnit testing framework effectively.



Benefits

Achieved reliable results with each test run.

Common Pitfalls in Java Testing

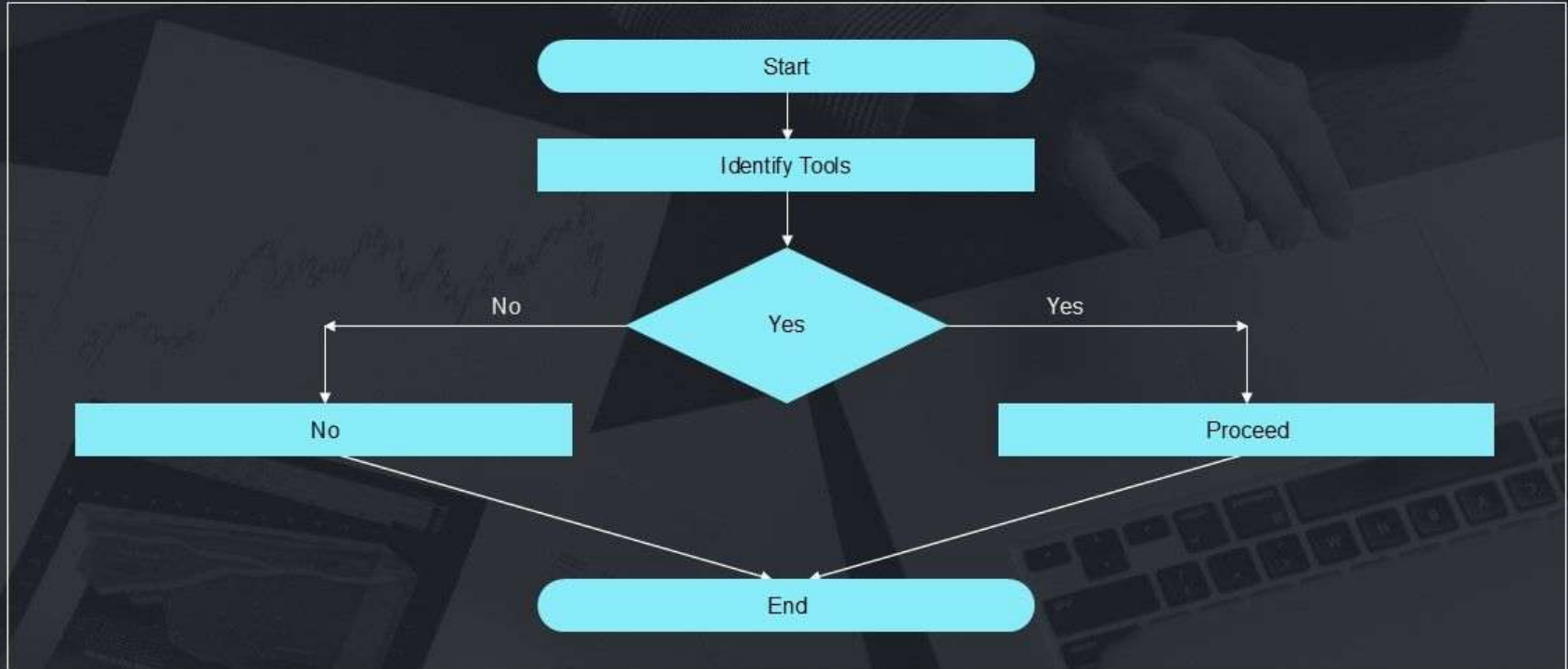


Best Practices for Writing Tests



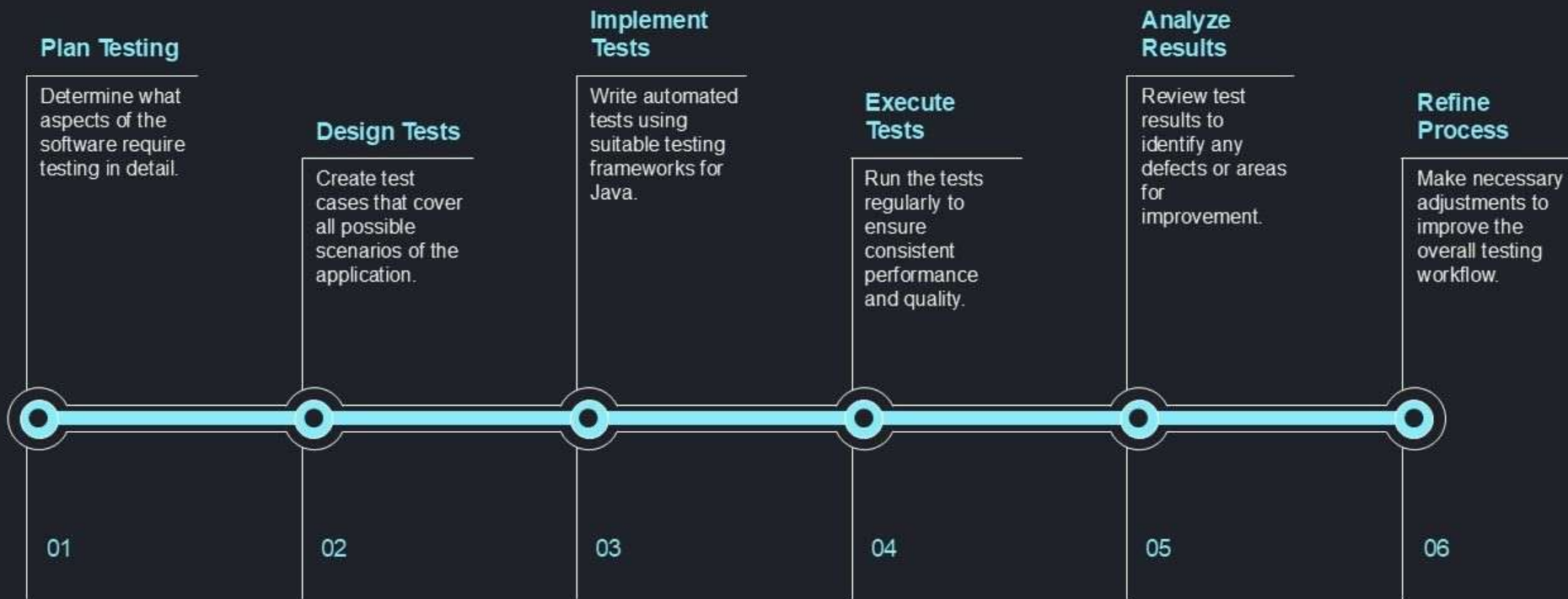
Test Automation Strategies with Java

Flow Chart



This is a sample flowchart for this slide. Please rearrange the flowchart to convey your message.

Integrating Testing into Agile Workflows



Future Trends in Java Testing Frameworks

AI Integration

Incorporating Artificial Intelligence into Java testing frameworks aims to enhance test case generation, enabling automated analysis of results and facilitating predictions of potential defects.

Cloud-Based Testing

The shift to cloud-based Java testing frameworks will enable teams to execute tests in scalable environments, reducing infrastructure costs and ensuring better collaboration across distributed teams.

Continuous Testing

Emphasizing continuous testing practices within Java frameworks allows for real-time feedback on code quality, promoting agile methodologies and accelerating the release cycles of software projects.

Conclusion and Key Takeaways

Diverse testing frameworks

Java offers a variety of frameworks to meet different testing needs.

Integration is crucial

Proper integration of testing frameworks enhances overall code quality.

Automation increases efficiency

Automated tests save time and reduce errors in the development process.

Community support is essential

Strong community support helps in troubleshooting and improving frameworks.

JUnit

Explore advanced features of JUnit for effective testing strategies.

TestNG

Utilize TestNG annotations for enhanced test management capabilities.

Mockito

Implement Mockito for mocking dependencies in unit tests efficiently.

Cucumber

Adopt Cucumber for behavior-driven development and automated acceptance tests.

*Further Reading
and Resources*

Thank You !



Address

123 Java Lane, Springfield, IL



Contact Number

(555) 123-4567



Email Address

contact@javatestframeworks.com