



BITS Pilani
Pilani Campus



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

By Prof A R Rahman
CSIS Group WILP



BITS Pilani
Pilani Campus

Course Name: Introduction to DevOps
Course Code : CSI ZG514/SE ZG514

By Prof A R Rahman
CSIS Group WILP



CS - 6 Continuous build and code quality

Coverage



Continuous build and code quality

- Manage Dependencies
- Automate the process of assembling software components with build tools
- Use of Build Tools- Maven, Gradle
- Unit testing
- Enable Fast Reliable Automated Testing
- Setting up Automated Test Suite – Selenium
- Continuous code inspection - Code quality
- Code quality analysis tools- SonarQube



Management of dependencies in DevOps

- In the world of software development, managing dependencies and external libraries is a crucial aspect of the DevOps process.
- Effective management ensures smooth integration, minimizes conflicts, and maximizes efficiency.
- The best practices and strategies for handling dependencies and external libraries in DevOps.
- Whether you are a seasoned developer or just starting your journey to streamline your development process.



Management of dependencies in DevOps

- In today's fast-paced software development landscape, where collaboration and agility are key, managing dependencies and external libraries is vital for project success.
- Dependencies are the external components, modules, or libraries that your software relies on to function correctly.
- These dependencies can be open-source libraries, frameworks, or even internal components from other teams or departments within your organization.



Understanding Dependencies and External Libraries

- Dependencies and external libraries come in various forms, including:
- **Direct dependencies:** These are the libraries or components that your software directly requires to function.
- **Transitive dependencies:** These are the dependencies that your direct dependencies rely on.
- Managing transitive dependencies can be challenging, as they introduce complexity and potential conflicts.
- To ensure smooth development and deployment processes, it's essential to understand the relationships and requirements of these dependencies.



Choosing the Right Dependency Management System

- To effectively handle dependencies, choosing the right dependency management system is crucial.
- Popular dependency management tools include:
- **Maven:** Widely used for Java projects, Maven simplifies dependency management and builds automation.
- **npm:** The default package manager for Node.js, npm offers a vast ecosystem of open-source packages.
- **pip:** The package installer for Python, pip facilitates installing and managing Python libraries.



Choosing the Right Dependency Management System

- **Gradle:** Known for its flexibility and extensibility, Gradle is a popular choice for JVM-based projects.
- **Composer:** Designed for PHP projects, Composer makes managing dependencies a breeze.
- Each system has its strengths and caters to specific programming languages and project requirements.
- Consider factors like community support, ease of use, and compatibility when choosing a dependency management system.



Defining a Solid Versioning Strategy

- Versioning plays a critical role in managing dependencies effectively.
- A solid versioning strategy ensures stability, compatibility, and easy upgrades.
- Some commonly used versioning schemes include:
- **Semantic Versioning (SemVer):** Following a “major.minor.patch” format (e.g., 1.2.3), SemVer provides a clear indication of backward compatibility and introduces breaking changes.
- **Lockfiles:** Lockfiles record the exact versions of dependencies used during development, ensuring consistent builds across different environments.
- Whichever versioning strategy you choose, it’s essential to communicate and enforce it within your development team to maintain consistency and minimize conflicts.



Automated Build and Deployment Pipelines

- DevOps emphasizes automation and continuous integration/continuous delivery (CI/CD) practices.
- Implementing automated build and deployment pipelines simplifies the management of dependencies and libraries.
- These pipelines automate tasks such as dependency resolution, compilation, testing, and deployment, allowing developers to focus on writing code and minimizing manual errors.



Monitoring and Dependency Updates

- Dependencies and external libraries often release new versions with bug fixes, security patches, and new features.
- Monitoring for updates and regularly updating dependencies is crucial to maintain the health and security of your software.
- Automated tools and systems can help track updates and perform routine checks, making the process more manageable.



Security Considerations

- Managing dependencies involves considering security aspects.
- Open-source libraries, while beneficial, can introduce vulnerabilities.
- Regularly monitoring security advisories and conducting vulnerability scans helps identify and mitigate potential risks.
- Additionally, incorporating code reviews and security testing into your development process ensures that dependencies are thoroughly evaluated.



Documentation and Communication

- Clear documentation and effective communication are vital for managing dependencies.
- Document the purpose, usage, and configuration details of each dependency.
- Provide guidelines for updating dependencies, resolving conflicts, and adding new ones.
- Regularly communicate these guidelines to your development team to maintain consistency and avoid misunderstandings.



Dependency Management in Microservices Architecture

- In a microservices architecture, managing dependencies becomes more complex due to the distributed nature of the system.
- Each microservice may have its own set of dependencies and versions.
- Adopting a service mesh or utilizing containerization technologies like Docker can help manage dependencies efficiently in such architectures.



Testing and Quality Assurance

- Testing and quality assurance are integral parts of dependency management.
- Comprehensive test suites should cover both the core functionality of your software and the behavior of dependencies under different scenarios.
- Automated testing frameworks and tools, along with continuous integration, aid in efficient testing and identifying any issues caused by dependencies.



Dealing with Legacy Code and Technical Debt

- Legacy codebases often pose challenges when managing dependencies.
- Outdated dependencies, conflicting versions, and poor documentation can make the process daunting.
- It's crucial to prioritize addressing technical debt, regularly updating dependencies, and refactoring code to reduce complexity and improve maintainability.



The Role of Package Managers

- Package managers play a pivotal role in managing dependencies and external libraries.
- These tools automate the process of downloading, installing, and updating packages.
- They also help resolve version conflicts and ensure consistent builds across different environments.
- Familiarize yourself with the package manager specific to your programming language to leverage its capabilities fully.



Best Practices for Effective Dependency Management

- To summarize, here are some best practices for managing dependencies and external libraries in DevOps:
- Clearly define and communicate versioning strategies.
- Utilize automated build and deployment pipelines.
- Implement continuous integration and continuous delivery (CI/CD) practices.
- Regularly monitor and update dependencies.



Best Practices for Effective Dependency Management

- Prioritize security by monitoring for vulnerabilities and conducting security testing.
- Document dependencies and provide guidelines for their usage and updates.
- Test thoroughly and incorporate quality assurance practices.
- Address technical debt and legacy code issues.
- Leverage package managers to simplify dependency management.



Automate the process of assembling software components with build tools

- Automated Build Tool is a software that compiles the source code to machine code.
- Automation tools are used to automate the whole process of software build creation and the other related processes like packaging binary code and running the automated tests.
- These automation tools can be categorized into two types i.e. Build-Automation Utility and Build-Automation servers.
- Build automation utilities perform the task of generating build artifacts.
- Maven and Gradle come under this category of building automation tools.
- There are three types of Build Automation servers i.e. On-demand automation, Scheduled automation, and Triggered automation.

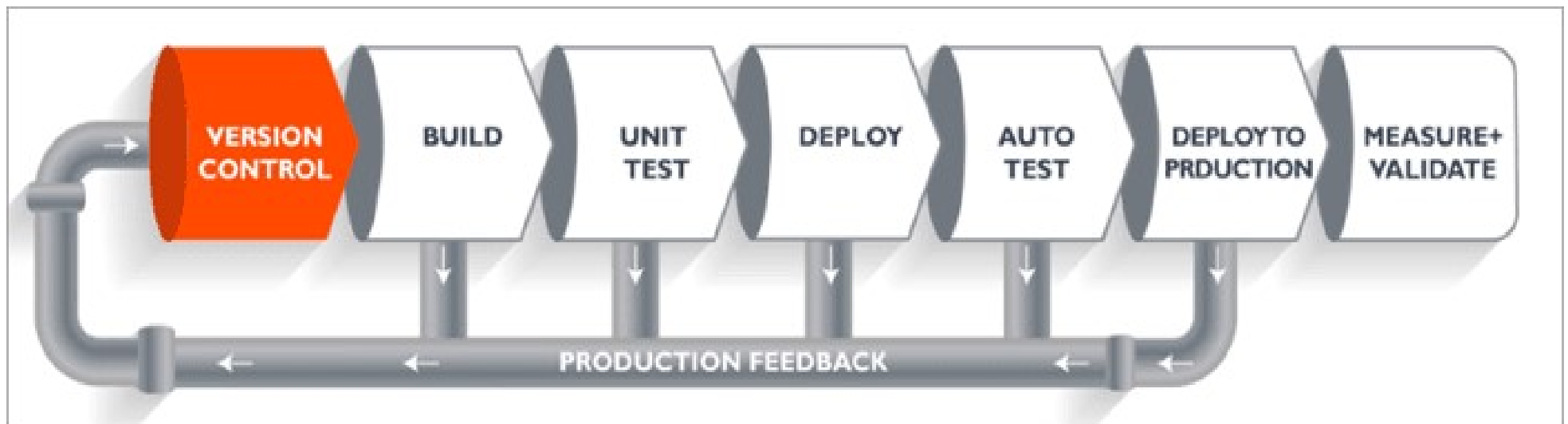


Automate the process of assembling software components with build tools

- Build Deployment and Continuous Integration Process
- If you want to implement Continuous Integration and Continuous Deployment then adopting the Build tool will be the first step of it.
- Build Tools provide the features of an extensive library of plugins, build & source code management functionalities, dependency management, parallel testing & build execution, and compatibility with IDE.

Automate the process of assembling software components with build tools

- Build Deployment and Continuous Integration Process





Automate the process of assembling software components with build tools

Challenges for Build Automation:

- #1) **Longer builds:** Longer builds take more time to run, it will increase the developer's wait time and thereby reduce productivity.
- #2) **Large volumes of builds:** If a large volume of builds is running, then you will get limited access to the build servers for that specific period.
- #3) **Complex builds:** Complex builds may require extensive manual efforts and may reduce flexibility.

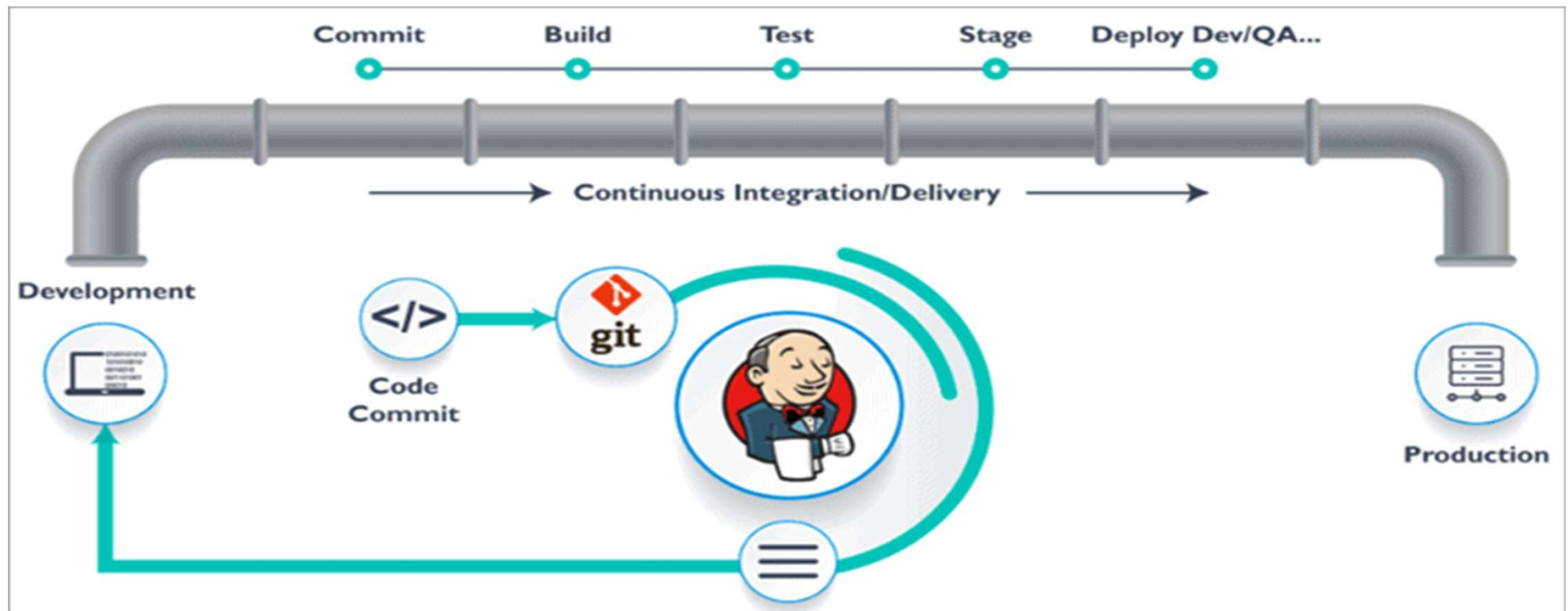
Based on your requirements you can look for features like integrations, pre-installed database services, or support for working on multiple projects.



Automate the process of assembling software components with build tools

- Benefits of Automation Build Tools
- Using the build automation software has several benefits as mentioned below:
- Saving time and money.
- Keeping a history of builds and releases. It will help in investigating the issue.
- Dependencies on key personnel will be eliminated through these tools.
- It will accelerate the process.
- It will perform redundant tasks.





Automate the process of assembling software components with build tools



Pro Tip: While selecting a build automation software the points to be considered include support for programming languages, support for multi-repo or mono-repo, and dependency management features.



Automate the process of assembling software components with build tools

Automation Tools	Best For	One Line Description	Free Trial	Price
 Jenkins	Jenkins Small to Large Businesses	Automation server used to Build, Deploy, and Automate any project.	No	Free
 Apache Maven Project	Maven Small to Large Businesses	Project management and comprehension tool.	No	Free
 Gradle Build Tool	Gradle Small to Large Businesses	Build Tool	30 days	Get a quote
 Travis CI	Travis CI Small to Large Businesses	Sync GitHub projects and test.	For 100 builds	Free for open source projects. Bootstrap: \$69/month Startup: \$129/month Small Business: \$249/month Premium: \$489/month
 Bamboo	Bamboo Small to Large Businesses	Continuous Integration & Deployment Build Server	30 days	Small Teams: \$10 for 10 jobs. Growing Teams: \$1100 for unlimited jobs.



Use of Build Tools- Maven, Gradle

- The Importance of Build and Package Manager Tools:
- Build and package manager tools simplify and automate the build process, making it easier for developers to manage dependencies, compile source code, and create deployable artifacts.
- These tools enhance collaboration, ensure consistency across environments, and enable seamless integration with continuous integration and deployment (CI/CD) pipelines.



Use of Build Tools- Maven, Gradle





Use of Build Tools- Maven, Gradle

- **Maven: Streamlining Java Build and Dependency Management:**
- Maven is a widely adopted build and package manager tool, particularly in the Java ecosystem.
- It utilizes a declarative approach, employing an XML-based Project Object Model (POM) to define project configuration, dependencies, and build processes.
- Maven provides a standardized build lifecycle, simplifying project management and promoting consistent builds.



Use of Build Tools- Maven, Gradle

- **Maven: Streamlining Java Build and Dependency Management:**
- With Maven, developers can effortlessly manage project dependencies, allowing the automatic resolution of libraries and frameworks from remote repositories.
- The tool facilitates the compilation of source code, execution of tests, packaging of artifacts, and deployment to remote repositories or application servers.
- Maven promotes code reuse, scalability, and faster development cycles.



Use of Build Tools- Maven, Gradle

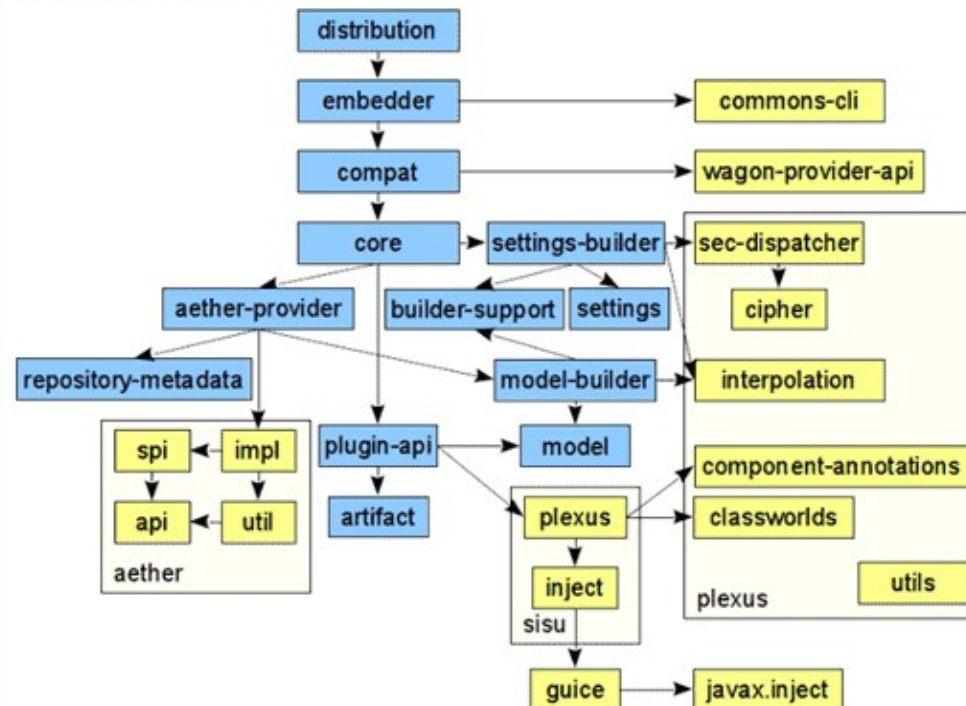
- Best for small to large businesses
Price: Free
- Maven is an application that provides functionalities for project management.
- It has functionalities for project building, reporting, and documentation.
- You will be able to access the new features instantly.
- It is extensible through plugins.
- There will be no limitation on building the number of projects into a JAR, WAR, etc.



Use of Build Tools- Maven, Gradle

- **Features:** It supports working on multiple projects simultaneously.
- There will be consistent usage for all projects.
- It has features for dependency management.
- It provides a large and growing repository of libraries and metadata.
- It provides functionality for release management: It can distribute individual outputs.
- For managing the releases and distributing the publications, Maven will be integrated with your system.
- No additional configuration will be required for this.

Use of Build Tools- Maven, Gradle





Use of Build Tools- Maven, Gradle

- The tool is good for building automation and dependency management.
- For dependency management, it provides support to the central repository of JARs.
- <https://maven.apache.org/>



Use of Build Tools- Maven, Gradle





Use of Build Tools- Maven, Gradle

- Gradle: Flexibility and Performance in Build Automation:
- Gradle is a powerful and flexible build automation tool that has gained significant popularity.
- It offers a Groovy or Kotlin-based Domain-Specific Language (DSL) for defining build scripts.
- Unlike Maven's XML-based configuration, Gradle provides a more concise and expressive syntax.



Use of Build Tools- Maven, Gradle

- Gradle: Flexibility and Performance in Build Automation:
- Gradle excels in managing complex build processes and supports a wide range of programming languages and platforms.
- It offers advanced features such as incremental builds, parallel execution, and caching, enhancing build performance.
- Gradle's extensive plugin ecosystem allows developers to extend its functionality and integrate with various tools and frameworks.



Use of Build Tools- Maven, Gradle

- Best for small to large businesses.
- Price: Gradle offers a free trial of 30 days for Gradle Enterprise.
- You can contact the company for the pricing of Enterprise subscriptions.
- Website: <https://gradle.org/>
- it has good integration capabilities.
- Gradle has features of web-based build visualization, collaborative debugging, parallel execution, incremental builds, task timeouts, etc.



Use of Build Tools- Maven, Gradle

- Gradle can be used for multiple project types i.e. mobile apps to microservices.
- It has functionalities for building, automating, and delivering software.
- It is an open-source platform. For dependency management, it provides the functionalities like transitive dependencies, custom dependency scopes, file-based dependencies, etc.



Use of Build Tools- Maven, Gradle

- Features:
- For software development, it will allow you to use any programming language.
- It can deploy on any platform.
- It supports monorepos as well as multi-repo strategy.
- It will help you to deliver continuously.
- It has various execution options like Continuous build, Composite Builds, Task Exclusion, Dry Run, etc

Use of Build Tools- Maven, Gradle

Build Scan

commons-lang3 build Sep 21, 2017 11:01:48 AM MST

10 dependencies resolved in 1 project across 3 configurations

- compileClasspath - 0.000s
- testCompileClasspath - 0.006s
 - junit:junit:4.12
 - org.apache.bcel:bcel:6.0
 - org.easymock:easymock:3.4
 - org.hamcrest:hamcrest-all:1.3
 - org.openjdk.jmh:jmh-core:1.17.4
 - net.sf.jopt-simple:jopt-simple:4.6
 - org.apache.commons:commons-math3:3.2
 - org.openjdk.jmh:jmh-generator-annprocess:1.17.4
- testRuntimeClasspath - 0.002s

Dependencies Required By

org.openjdk.jmh:jmh-core:1.17.4

net.sf.jopt-simple:jopt-simple:4.6

org.apache.commons:commons-math3:3.2

Home > Dependencies

Close dependency details (esc)



Unit Testing

- What Is Unit Testing?
- A unit test is a type of software test that focuses on testing individual components of a software product.
- Software developers and sometimes QA staff write unit tests during the development process.
- The 'units' in a unit test can be functions, procedures, methods, objects, or other entities in an application's source code.



Unit Testing

- Each development team decides what unit is most suitable for understanding and testing their system.
- For example, object-oriented design tends to treat a class as the unit.
- Unit testing relies on mock objects to simulate other parts of code, or integrated systems, to ensure tests remain simple and predictable.
- The main goal of unit testing is to ensure that each unit of the software performs as intended and meets requirements.
- Unit tests help make sure that software is working as expected before it is released.



Unit Testing

- Unit testing refers to a software development practice in which you test each unit of an application separately.
- In this scenario, a unit could refer to a function, procedure, class, or module — essentially, it's the smallest testable part of the software.
- Unit testing ensures it works as it should before the entire system is integrated.
- It's vital to have unit tests that run speedily, are isolated from external dependencies, and are easy to automate for accuracy and convenience.



Unit Testing

- Developers could choose to manually write and execute their own unit test cases, which is often ideal for smaller projects or situations that call for more hands-on examination of the code.
- However, automation is generally preferred to ensure that unit tests run efficiently, consistently, and at scale.
- Automated testing frameworks like JUnit, NUnit, and PyTest are commonly used to streamline this process.



When Should Unit Testing Be Performed, and by Whom?

- Unit testing is typically the first level of software testing and is performed before integration testing, acceptance testing, and system testing.
- This helps identify any issues with the codebase before too much time is invested in building the full features.
- Developers will typically write the unit tests, as they're the ones who know best how any individual class or function should work.
- A lot of the time, they'll run these tests themselves since each test takes a negligible amount of time.



When Should Unit Testing Be Performed, and by Whom?

- However, in some cases, they'll opt to hand it over to the Quality Assurance (QA) process team instead.
- In general, unit testing can be handled by any team member with access to the software's source code and a good understanding of its structure and functionality.



Unit Testing

- The main steps for carrying out unit tests are:
- Planning and setting up the environment
- Writing the test cases and scripts
- Executing test cases using a testing framework
- Analyzing the results



Unit Testing

- Some advantages of unit testing include:
- Early detection of problems in the development cycle
- Reduced cost
- Test-driven development
- More frequent releases



Unit Testing

- Some advantages of unit testing include:
- Enables easier code refactoring
- Detects changes which may break a design contract
- Reduced uncertainty
- Documentation of system behavior



Unit Testing

- **Benefits of Unit Testing**
- As the first layer of testing, performing unit tests is key to building and delivering a robust software product.
- Simply put, if your individual units aren't working as they should, they certainly won't work together. The benefits of unit testing include:
 - **1. Better code writing**
 - Unit testing, by nature, calls for each component of the software to have a properly defined responsibility that you can test in isolation.
 - This motivates developers to write high-quality code that's easy to maintain.



Unit Testing

- **Benefits of Unit Testing**
- As the first layer of testing, performing unit tests is key to building and delivering a robust software product.
- Simply put, if your individual units aren't working as they should, they certainly won't work together. The benefits of unit testing include:
 - **1. Better code writing**
 - Unit testing, by nature, calls for each component of the software to have a properly defined responsibility that you can test in isolation.
 - This motivates developers to write high-quality code that's easy to maintain.



Unit Testing

- **2. Early bug detection**
- Running unit tests helps detect bugs early in software development and pinpoint exactly where the bug lies.
- This allows you to fix it faster and avoid bigger problems later when dependencies among different software product components become more complex.
- **3. Documented results**
- Unit tests are ideal for chalking out your software's logic, as they demonstrate exactly what behavior you expect from each component.
- This is great for knowledge transfer, regression prevention, and as a standard for future software products you develop.



Unit Testing

- **4. Less need for regression testing**
- Regression testing involves retesting the software as a whole for functionality and reliability after changes are incorporated.
- This can be time-consuming and expensive. However, with unit testing, functionality is verified from the get-go, making regression tests much shorter and easier to run.
- **5. Better overall development process**
- Businesses that incorporate unit testing enjoy a more robust development lifecycle, one that fixes issues as early as possible.
- Moreover, developers are motivated to write code that can be repeatedly run without difficulty, making for a more agile coding process.



Anatomy of a Unit Test

- There are five main aspects of a unit test:
- 1. Test fixtures
- Also known as test context, test fixtures are the components of a test case that create the initial conditions for executing the test in a controlled fashion.
- These ensure that you have a consistent environment to repeat your testing in (such as configuration settings, user account, sandbox environment, etc.), which is important when you're testing the same feature over and over to get it just right.



Anatomy of a Unit Test

- **2. Test case**
 - This is a piece of code that determines the behavior of another piece of code. Developers need to define exactly what they expect from any unit in terms of results — the test case ensures that the unit produces exactly those results.
- **3. Test runner**
 - This is a framework that enables the execution of multiple tests at the same time by quickly scanning your directories or codebase to file and execute the right tests.
 - A test runner can also run tests by priority, manage the test environment to be free of any external dependencies, and provide you with a core analysis of the test results.



Anatomy of a Unit Test

- 4. Test data
- This refers to the data you select to run a test on your chosen unit.
- The goal is to choose data that covers as many possible scenarios and outcomes for that unit as possible. Common examples include:
- **Normal cases:** regular input values within acceptable limits
- **Boundary cases:** values at the boundaries of the acceptable limits
- **Corner cases:** values that represent extreme or unusual scenarios that could affect your unit or even your whole system
- **Invalid/error cases:** input values that fall outside the valid range, used to assess how the unit responds, including any error handling or messages



Anatomy of a Unit Test

- 5. Mocking and stubbing
- In most unit tests, developers focus on the specific unit.
- However, in some cases, your test will call for two units, especially if there are any necessary dependencies.
- Mocking and stubbing serve as substitutes for those dependencies so that your unit can still be tested in isolation.
- For instance, you might have a 'user' class that depends on an external 'email sender' class for delivering email notifications.
- In that case, developers can create a mock object of the 'email sender class' to test the behavior of the 'user' class without actually sending anybody any emails.



How to Do a Unit Test

- The basic procedure for running a unit test involves the following steps:
- **1. Identify the unit to test**
- Decide which specific code unit you'll be testing: a method, a class, a function, or anything else.
- Study the code, decide on the logic needed to test it, and list the test cases you plan to cover.
- **2. Use high-quality test data**
- You should always run your tests with data as similar to what the software product will work with in real life as possible.
- Be sure to cover edge cases and invalid data as well to see how your units function under those circumstances.
- Also, avoid hard-coding data into your test cases, as this makes them harder to maintain.



How to Do a Unit Test

- 3. Choose between manual and automated testing
- As the name suggests, manual testing requires developers to manually run the code to see if your unit is behaving as expected.
- For automated testing, they'll write a script that automates the code interactions.
- Both have their uses — automated testing lets you cover more ground faster, but manual testing is a more hands-on option for situations that require a more creative or intuitive perspective.



How to Do a Unit Test

- **4. Prepare your test environment**
 - This involves preparing your test data, any mock objects, and all the necessary configurations and preconditions for unit testing.
 - Ideally, you'll also want to isolate the code in a dedicated testing environment to keep the test free of any external dependencies.
- **5. Write and run the test**
 - If you're using an automated testing approach, start by writing a script for a test runner.
 - You can create test cases before you write the actual code.
 - This will help you keep any gaps in logic or software requirements before you invest time and effort in writing the code.
 - Then, run your tests. Cover all the test cases you listed in the previous steps. Ensure you reset the test conditions before each run of your unit test, and try to avoid any dynamically generated data that could negatively affect test results.



How to Do a Unit Test

- **4. Prepare your test environment**
 - This involves preparing your test data, any mock objects, and all the necessary configurations and preconditions for unit testing.
 - Ideally, you'll also want to isolate the code in a dedicated testing environment to keep the test free of any external dependencies.
- **5. Write and run the test**
 - If you're using an automated testing approach, start by writing a script for a test runner.
 - You can create test cases before you write the actual code.
 - This will help you keep any gaps in logic or software requirements before you invest time and effort in writing the code.
 - Then, run your tests. Cover all the test cases you listed in the previous steps. Ensure you reset the test conditions before each run of your unit test, and try to avoid any dynamically generated data that could negatively affect test results.



How to Do a Unit Test

- 6. Evaluate your result and make any fixes required
- Wherever your tests fail, evaluate where the problem lies and tweak the code.
- Then, the tests will be rerun to verify that the new code has solved the problem.
- This could take some time, which is why it's necessary to account for a buffer period in the software development lifecycle.



Unit Testing Types

- Black box testing
- This form of unit test focuses on the unit's external functionality and behavior, ignoring its internal structure.
- Your team will draft test cases based on the unit specifications and the expected inputs and outputs.
- For instance, they might test a login function by inputting different sets of valid and invalid credentials to see if it behaves correctly.



Unit Testing Types

- 2. White box testing
- If you want to consider the internal structure and implementation, with test cases designed to cover all the code branches and segments in the unit, consider white box testing.
- This includes testing all possible execution paths in the code, such as each branch of an if-else statement, to ensure every possible condition is tested and behaves as expected.



Unit Testing Types

- 3. Grey box testing
- This is also known as semi-transparent testing. In this case, the testers only have partial awareness of the unit's internal details.
- It includes pattern testing, orthogonal pattern testing, matrix testing, and regression testing.
- In a unit test example, you can use partial knowledge of the database schema to test how the system handles specific query inputs.



Unit Testing Types

- 4. Code coverage testing
- Code coverage testing involves measuring the extent to which the code has been tested by techniques such as statement coverage, decision coverage, and branch coverage.
- This helps identify untested code sections and increases the thoroughness of your unit tests. For instance, you could run tests to ensure every line of code in a function has been executed at least once.



The Unit Testing Life Cycle

- Review the code written after you implement your test.
- Refactor the test and make suitable changes to the code based on the insights you received about what's happening with it.
- Execute the test with suitable input values and compare the actual results with the expected ones.
- Fix any bugs detected during the testing process. This helps you prepare code that's as clean as possible before you send it into production.
- Re-execute your tests to verify the results after you've made the changes. This helps you keep track of everything you have done and ensure that the changes haven't led to regressions in the existing functionality.



Enable Fast Reliable Automated Testing

- Software teams are embracing automated testing to improve quality and speed up development.
- What was once a nice-to-have practice has become essential for delivering reliable software quickly.
- Let's explore what automated testing really means for development teams and how it shapes the way we build software today.



Why Automate Your Testing?

- The benefits of automated testing go far beyond just saving time.
- When tests run automatically, teams can catch bugs within minutes instead of days.
- Unlike manual testing, automated tests run the exact same way every time, removing inconsistencies from human testers.
- This quick, reliable feedback helps developers fix issues right when they appear.



Why Automate Your Testing?

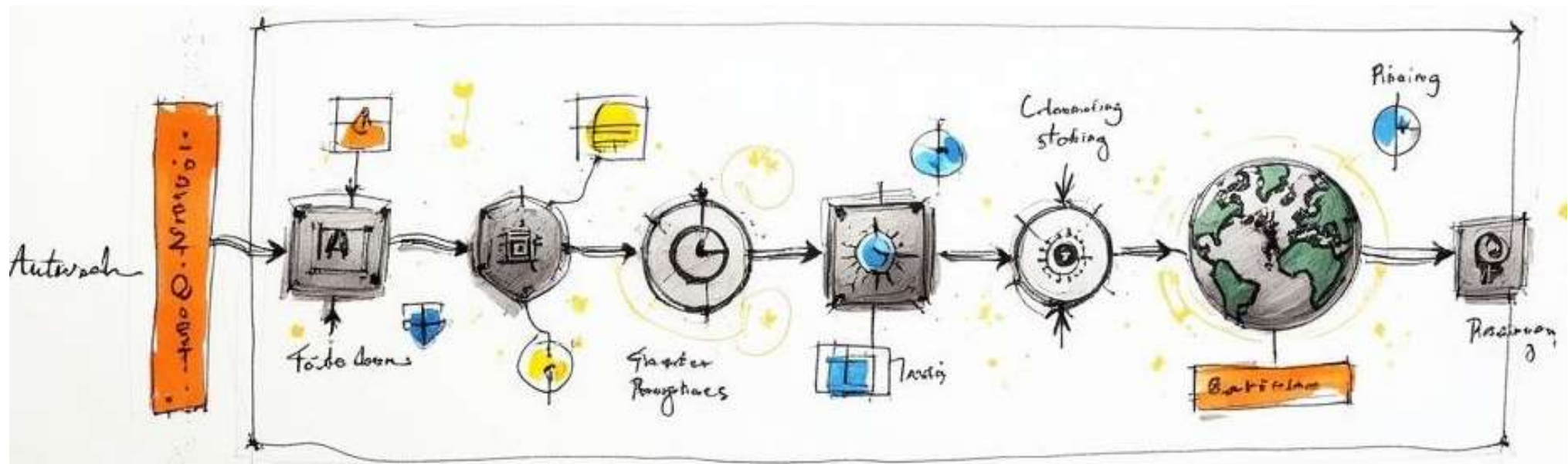
- While setting up test automation requires upfront work, it pays off many times over.
- Teams spend less time running repetitive tests and more time building features.
- The consistent quality checks also mean fewer bugs make it to production.
- Most importantly, automated testing is a key piece of **Continuous Integration/Continuous Delivery (CI/CD)**, letting teams ship updates safely and frequently.



Measuring the Impact of Automation

- To understand if automation is working, teams need clear metrics.
- The most telling **Key Performance Indicators (KPIs)** include **defect detection rate** (how many bugs we catch), **test execution time** (how fast tests run), and **test coverage** (how much code gets tested).
- For example, when automated tests catch more bugs before release while running faster each month, that shows the investment is paying off.
- The shift toward automation is clear across the industry.
- A recent study found that **44%** of IT teams now automate at least **50%** of their testing - a big change from just a few years ago.

Measuring the Impact of Automation





Building a Successful Automation Strategy

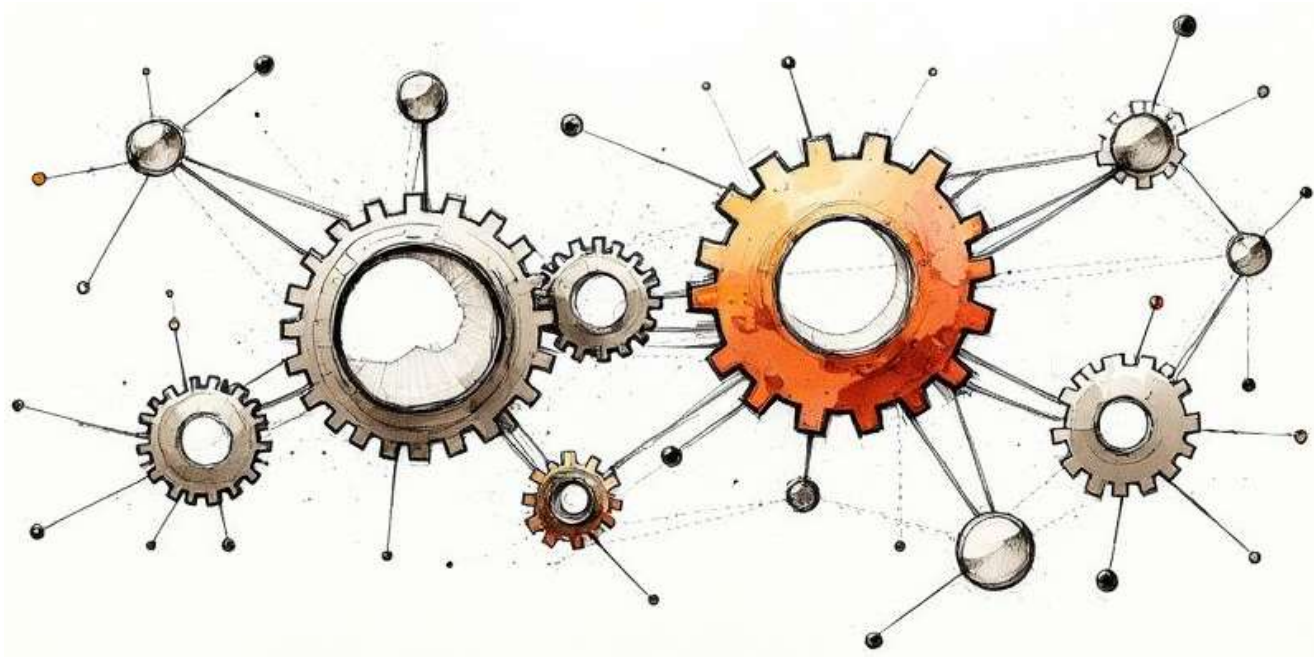
- Creating an effective test automation approach takes careful planning.
- Start by identifying which tests will give the biggest return - usually the ones that are repetitive, time-consuming, or prone to human error.
- Pick testing tools that match your team's skills and needs.
- A good framework makes it easier to write and maintain tests over time.



Finding Your Perfect Balance: Manual vs. Automated Testing

- Make automation part of your regular development process, from unit tests to end-to-end testing.
- Keep evaluating what works and what doesn't - successful automation is an ongoing journey of improvements.
- With the right strategy, automated testing becomes a powerful tool for delivering better software faster.

Finding Your Perfect Balance: Manual vs. Automated Testing





Finding Your Perfect Balance: Manual vs. Automated Testing

- Choosing between manual and automated testing involves understanding each approach's unique strengths and limitations.
- The goal isn't to completely replace manual testing, but to smartly combine both methods for maximum impact.
- Just like having the right tools for different tasks, each testing method has specific jobs it does best.



Finding Your Perfect Balance: Manual vs. Automated Testing

- When to Automate
- Automated tests excel at repetitive work that would be tedious and time-consuming for humans.
- For instance, regression testing - checking if existing features still work after code changes - is perfect for automation.
- Running the same tests hundreds or thousands of times is where automated testing shines.
- A real example is testing an e-commerce checkout: While a human tester could only check a few orders at a time, automated tests can simulate hundreds of simultaneous purchases to spot performance issues.



Finding Your Perfect Balance: Manual vs. Automated Testing

- The Power of Human Insight
- Some testing scenarios need a human touch.
- Take usability testing - only real people can effectively judge how intuitive and user-friendly an interface feels.
- Similarly, exploratory testing requires creative thinking and intuition that automated tools simply can't replicate.
- Human testers bring unique problem-solving skills and can spot potential issues that might slip past automated checks.



Finding Your Perfect Balance: Manual vs. Automated Testing

- Striking the Right Balance
- Many successful teams use what's called **hybrid testing** - mixing automated and manual approaches based on their strengths.
- According to recent data, **46%** of companies now automate at least half of their testing work, showing a clear trend toward balanced testing strategies.
- The key is using automation for repetitive tasks while keeping human testers focused on areas that need their judgment and creativity.



Finding Your Perfect Balance: Manual vs. Automated Testing

- Building Your Hybrid Strategy
- Creating an effective hybrid testing approach takes careful planning.
- Start by looking at your current testing process and identify which tasks eat up the most time or are prone to human error - these are prime candidates for automation.
- Choose testing tools that match your team's skills and tech stack.
- Keep track of what's working and adjust your approach based on results and feedback.
- How to Master Sitemaps. Regular reviews help ensure your testing strategy stays effective as your project grows and changes.



Key Performance Indicators for Test Automation

- These essential metrics help teams understand if their test automation is working well:
- **Defect Detection Rate:** Shows what percentage of total bugs your automated tests catch.
- When this number is high, it means your tests are spotting problems before users do.
- **Test Execution Time:** The speed of your test runs affects how often you can test.
- Faster tests mean quicker feedback for developers.



Key Performance Indicators for Test Automation

- **Test Coverage:** Measures how much of your code the tests check.
- While **100%** coverage isn't always needed, you want good coverage of your core features.
- **Automation Maintenance Cost:** The time your team spends fixing and updating tests.
- Well-designed tests need less maintenance.
- **Time Saved by Automation:** Compare the hours spent on manual vs automated testing to show the real benefits.
- This helps prove the value of investing in automation.



Key Performance Indicators for Test Automation

- When teams focus on these metrics, they often see clear improvements.
- For example, many companies find their automated tests catch bugs much earlier in development after improving their detection rates.
- Better testing directly leads to better software quality and faster releases.



Building a Measurement Framework

- To get the most from these metrics, set up a clear system to track them:
- **Define clear goals:** Pick specific targets like "cut test runtime in half" or "catch 20% more bugs before release"
- **Select relevant KPIs:** Choose the metrics that match your goals and give useful insights
- **Track and analyze data:** Regularly check your metrics to spot trends and areas for improvement. Use this data to show your team's progress.
- With good measurement in place, test automation becomes more than just running scripts - it becomes a key part of delivering quality software efficiently.
- This focus on data helps you improve your testing approach, show its value to others, and achieve better results.



Key Considerations for Framework Design

- Your test automation framework needs several essential elements to succeed over time:
- **Modularity:** Think of your tests like building blocks - break them into smaller, focused pieces that are easy to understand and reuse.
- Just as you'd build with LEGOs, each test component should have a clear purpose while fitting into the larger structure.
- **Abstraction:** Keep test logic separate from implementation details.
- For example, if you abstract UI interactions, you can update interface elements without rewriting all your tests.



Key Considerations for Framework Design

- **Data-Driven Testing:** Store test data separately from test code.
- This lets you run the same test with different inputs - like using a template where you only change the data while keeping the structure intact.
- **Reporting and Logging:** Create clear reports showing test results, failures and patterns.
- Good logs help quickly identify and fix issues when tests fail.



Essential Components of a Robust Framework

- A strong framework needs these key parts working together:
- **Test Data Management:** Have a clear strategy for handling test data from files, databases, or APIs.
- This keeps data consistent and reduces dependencies between tests.
- **Environment Management:** Set up test environments automatically to ensure reliable testing conditions without manual work.
- **Library of Reusable Functions:** Build a collection of common test operations that you can use across different tests.
- This makes maintenance easier and keeps code clean.



Avoiding Common Pitfalls

- Watch out for these frequent issues when building your framework:
- **Over-Engineering:** Keep things simple and focused on what you actually need.
- Extra complexity just makes maintenance harder.
- **Lack of Documentation:** Write clear docs explaining how the framework works and how to use it.
- This helps the whole team contribute effectively.



Avoiding Common Pitfalls

- **Ignoring Test Maintenance:** Set aside time to update dependencies, clean up code, and handle technical debt regularly.
- Following these practices helps create a framework that truly serves your team's needs and grows with your application.
- A solid framework speeds up testing, improves coverage, and catches more bugs before they reach production - leading to better quality software and faster development overall.



Setting up Automated Test Suite – Selenium

- Setting up a Selenium environment is a straightforward process that can unlock powerful automated testing for web applications.
- Let's walk through how to get started with Selenium, ensuring we cover every essential step.
- You'll have everything you need to get Selenium working efficiently, so you can focus on testing and automating your web tasks!



Setting up Automated Test Suite – Selenium

- Step 1: Install the Necessary Packages
- The first step is to ensure you have Selenium and a compatible web driver.
- We'll use Python as the base for our setup, but Selenium works with other languages too, like Java and C#.
- However, Python's simplicity makes it a good choice.



Setting up Automated Test Suite – Selenium

- Install Selenium

First, install Selenium using pip.

- Open your terminal (or command prompt) and run the following:
- `pip install selenium`
- This installs Selenium, giving you the tools to interact with and automate web browsers.



Setting up Automated Test Suite – Selenium

- 2. Install a Web Driver

Selenium works by controlling a web browser through a driver.

- Each browser has its driver. For example:
- Chrome: ChromeDriver
- Firefox: [GeckoDriver](#)
- Edge: [Edge WebDriver](#)
- For simplicity, let's assume we're using Chrome.
- Download ChromeDriver and place it in a known directory (such as C:/webdrivers/ or /usr/local/bin/).



Setting up Automated Test Suite – Selenium

- Step 2: Verify the Installation
- Now that we've installed Selenium and ChromeDriver, let's write a small script to verify that everything is working.
- Here's a Python code snippet to open Chrome, navigate to a website, and confirm that Selenium is properly controlling the browser:



Setting up Automated Test Suite – Selenium

```
from selenium import webdriver

# Specify the path to the ChromeDriver you downloaded
chrome_driver_path = 'C:/webdrivers/chromedriver.exe' # Update this to your actual path

# Initialize the WebDriver
driver = webdriver.Chrome(executable_path=chrome_driver_path)

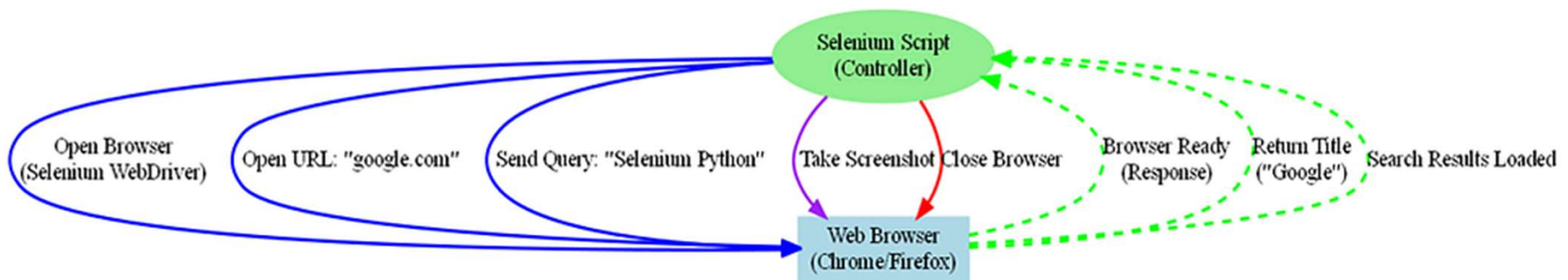
# Open a website
driver.get("https://www.google.com")

# Print the page title to confirm the connection
print(driver.title)

# Close the browser after 5 seconds
driver.implicitly_wait(5)
driver.quit()
```

Setting up Automated Test Suite – Selenium

- An interaction diagram between the script and the web browser is shown in figure 1.
- Here, the browser receives commands from your script (acting as the controller) and then responds accordingly (i.e., opening the specified URL).





Setting up Automated Test Suite – Selenium

Let's walk through the edges of the diagram step by step:

- **Blue Edges:** Think of these as the “commands” your Selenium script sends to the browser.
- For example, when the script tells the browser to open, or navigate to a specific URL like “google.com,” it's the blue edge at work.
- It's basically the “action request” coming from the controller.



Setting up Automated Test Suite – Selenium

- **Green Dashed Edges:** Now, after every command, the browser needs to respond, right? That's where the green dashed edges come in.
- These edges show how the browser “talks back” to the script, letting it know things like, “The page is loaded,” or “I’m ready for the next command.”
- **Purple Edge:** This one's a bit special — it's for actions that stand out, like taking a screenshot.
- It's like a milestone where the script pauses to capture what's on the screen before moving forward.



Setting up Automated Test Suite – Selenium

- **Red Edge:** Finally, when it's all done, the red edge steps in.
- It represents the closing of the browser, marking the end of the interaction between the script and the browser.
- It's like a polite “goodbye” after the job is finished.

This flow keeps the interaction smooth and easy to follow, and each edge color makes it clear what's happening at every step.



Setting up Automated Test Suite – Selenium

- Step 3: Automating Basic Interactions
- Now that your Selenium environment is set up, let's go one step further and interact with the web page. We'll create a script that automates a simple task
- Here's how the code would look like:



Setting up Automated Test Suite – Selenium

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

# Initialize the WebDriver and open Chrome
driver = webdriver.Chrome(executable_path=chrome_driver_path)

# Open Google's homepage
driver.get("https://www.google.com")

# Find the search box using its name attribute value
search_box = driver.find_element_by_name("q")

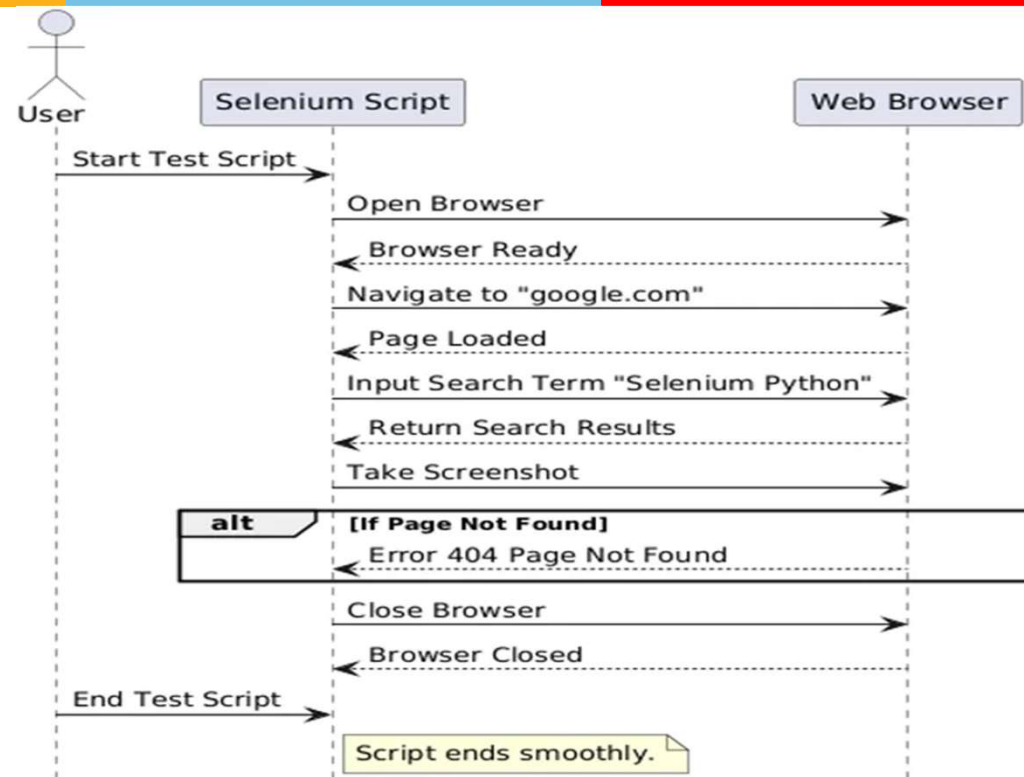
# Send search query and simulate hitting the "Enter" key
search_box.send_keys("Selenium Python")
search_box.send_keys(Keys.RETURN)

# Let the results load
driver.implicitly_wait(5)

# Print the title of the page after search
print(driver.title)

# Close the browser
driver.quit()
```

Setting up Automated Test Suite – Selenium



A sequence diagram showing how different interactions happen step by step is shown in figure 2: from opening the browser, inputting the search term, and retrieving the results, to closing the browser.



Here's how the interaction flows between the user, the Selenium script, and the web browser:

- First, the **user kicks things off** by starting the script.
- The moment this happens, the **script opens the browser**, which is like setting the stage for everything else.
- Next, the **browser gets to work**. It navigates to the URL you've specified, and once the page is fully loaded, it returns the search results.
- It's like the browser saying, "Here's what you asked for!"
- Now, the script takes a bit of a snapshot literally! It **captures a screenshot** of the loaded page to preserve that moment.



Here's how the interaction flows between the user, the Selenium script, and the web browser:

- But what if things don't go as planned? Let's say the page doesn't exist or there's an error.
- No worries the script is prepared for that too.
- It gracefully handles the situation, showing a **404 error** message if the page isn't found, so you always know what's going on.
- Simple, smooth, and a bit intuitive, right? Everything works together to make sure the interaction goes off without a hitch!



Setting up Automated Test Suite – Selenium

- Step 4: Handling Dynamic Content
- Web pages are often dynamic, meaning elements may load after the initial page load.
- To handle this, we use **explicit waits** to wait for certain conditions before interacting with elements.
- For example, we'll wait for search results to load before taking any action.
- Here's an example of explicit waits in action:



Setting up Automated Test Suite – Selenium

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Open Google and perform a search as before
driver = webdriver.Chrome(executable_path=chrome_driver_path)
driver.get("https://www.google.com")
search_box = driver.find_element_by_name("q")
search_box.send_keys("Selenium WebDriver")
search_box.send_keys(Keys.RETURN)

# Wait until the results are loaded
try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "search"))
    )
    print("Search results are ready.")
finally:
    driver.quit()
```

Setting up Automated Test Suite – Selenium

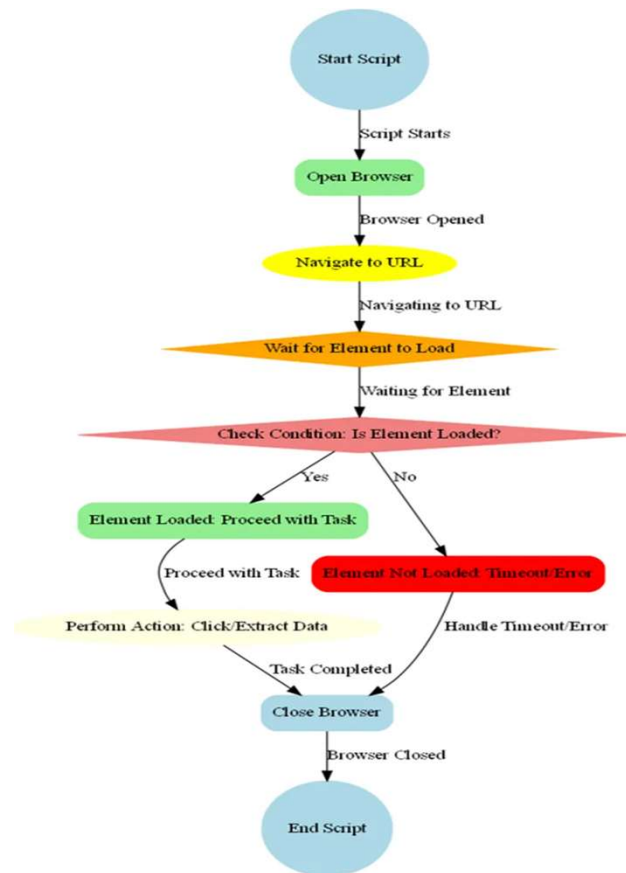


Figure 3 illustrates the logic flow: waiting for the element to load, checking the condition, and then proceeding with the task.



Setting up Automated Test Suite – Selenium

- Step 5: Incorporating Screenshots
- You can take screenshots at any point in your test to document the results or troubleshoot issues.
- Here's how you can add this functionality:
- # Take a screenshot of the Google homepage
`driver.save_screenshot('google_homepage.png')`
- # After performing a search, take another screenshot
`driver.save_screenshot('search_results.png')`

Setting up Automated Test Suite – Selenium

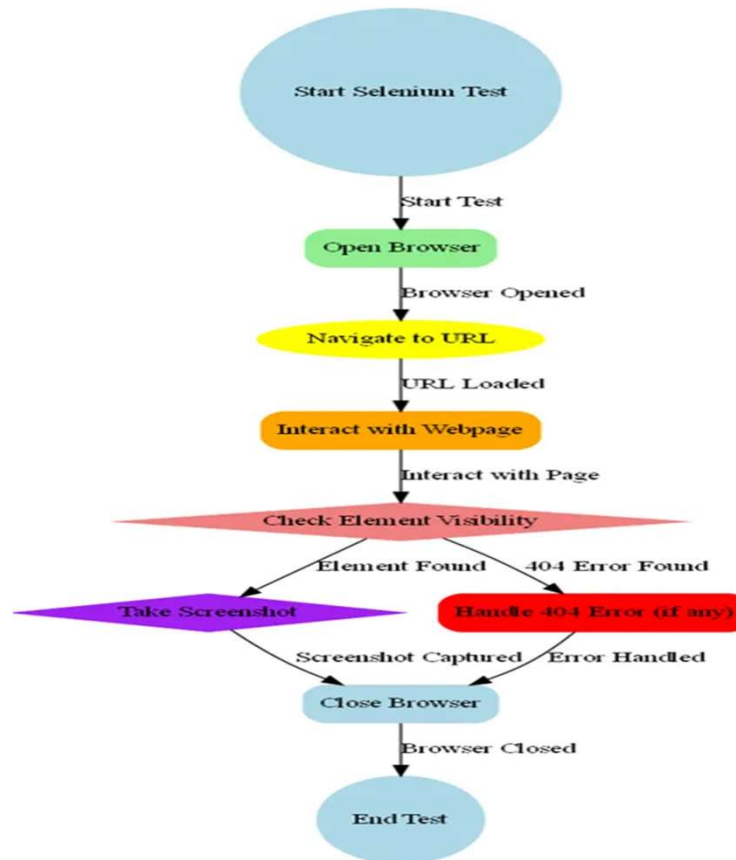


Figure 4 showcases the image flow from webpage interaction to screenshot capture to demonstrate how and when you take screenshots during a test.



Setting up Automated Test Suite – Selenium

- Step 6: Advanced Features — Headless Browsing
- Selenium allows you to run browsers in **headless mode**, which means no actual browser window will open. This is particularly useful for automation on servers or continuous integration pipelines where GUI interaction isn't necessary.
- To run Chrome in headless mode:
- from `selenium.webdriver.chrome.options` import `Options`

```
# Set Chrome options to run headlessly
```

```
chrome_options = Options()
```

```
chrome_options.add_argument("--headless")
```

```
# Initialize the WebDriver with the headless option
```

```
driver = webdriver.Chrome(executable_path=chrome_driver_path, options=chrome_options)
```


Setting up Automated Test Suite – Selenium

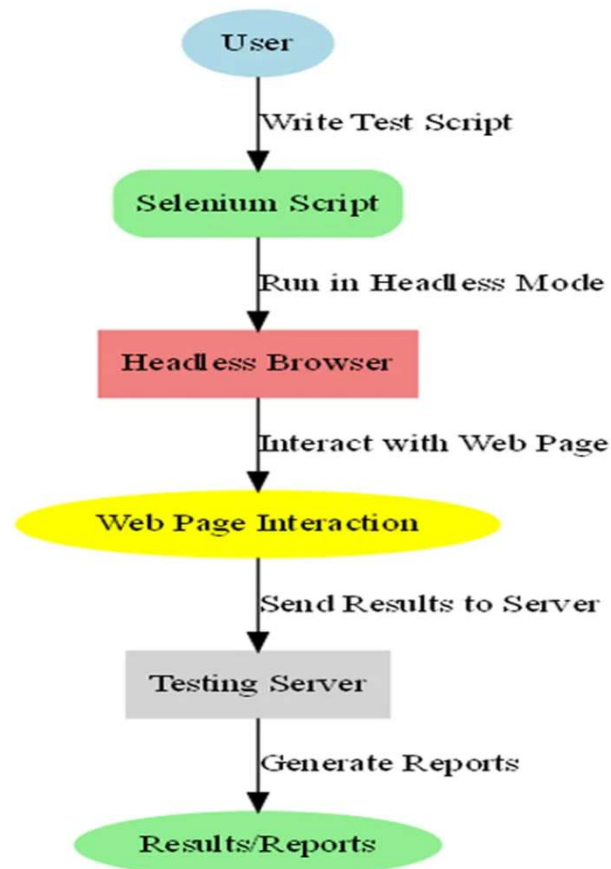


Figure 5 below represent an architecture to show how headless mode runs entirely in the background without any visible interface, useful for testing on servers.



Setting up Automated Test Suite – Selenium

- By now, you've set up Selenium, tested it with a browser, interacted with web pages, used dynamic waits, and even taken screenshots.
- With these steps, you're equipped to begin automating web testing for real-world scenarios.
- Having this environment in place can significantly streamline testing workflows.



Continuous code inspection - Code quality

- How to Define, Measure and Continuously Improve Code Quality?
- One of the basic but important customer expectations is – the software product should be of very good quality.
- That makes sense as well.
- However, what exactly “good quality” means?
- Here are characteristics of good quality software:
- Software should be able to perform all its functionality as expected (defined in acceptance criteria).



Continuous code inspection - Code quality

- Software should meet all non-functional “ilities” (like scalability, reliability etc.,).
- Code quality should be great with minimal technical debt.
- When it comes to code-quality, people in software world have different perceptions and interpretation around it.
- All these interpretations are abstract mostly and do not define code-quality term in measurable terms.
- “Code quality is an indicator of how quickly a developer can add business value to a software system”



Seven sins of code quality

Bugs and Potential Bugs – Bugs and Potential Bugs is the most urgent sin as it shows what's wrong in your code currently and what can go wrong tomorrow, e.g. `NullPointerException`.

Coding Standards Breach

Transgressors are too lazy to learn and follow your team's standards about whether or not to use spaces in if statements.

More serious example of this type of sin is the failure to follow naming conventions.



Seven sins of code quality

- Duplications

It doesn't seem like a big deal but it's not efficient in the long run.

- Similar to Murphy's Law, the more places a chunk of logic has been duplicated into, the more likely it is that it will need to be changed, probably with a high level of urgency or criticality

- Lack of Unit Tests

Unit tests help keep bugs and regressions from slipping into production code.

- And when you make a change to existing code, they help you know that you didn't break it.



Seven sins of code quality

- Bad Distribution of Complexity

It's okay for a program to have some complex files and methods.

- But if you have too many of those, then the next coder who has to work on the application will have a hard time understanding what's going on in the code.
- And if she has a hard time understanding it, she'll have an even harder time successfully modifying it.



Seven sins of code quality

- Not Enough or Too Many Comments

This is a measure of maintainability.

- It looks at how often you make the caller of your method look at the internal details (code) to understand what's happening, versus reading intentional documentation that (ideally) explains what should be passed in, what will be returned, and perhaps even what will happen in between.
- Comments are measured because they're part of what makes a system easy (or not) to work on.



Seven sins of code quality

- Spaghetti Design
- This is like having high complexity at the project architecture level, rather than in a single method or file.
- New developers on the team will have a hard time understanding how the project is organized, and where new code should be put.



How to measure Code Quality?

- Code quality needs to be measured in form of density.
- Abstract numbers are useless.
- For instance, 400 potential bugs as a number in itself doesn't tell the severity of the issue.
- However when one talks about it in form of potential bug density, you get to know better picture.
- Also it's important to observe the trend of code quality through a period of time.



How to measure Code Quality?

- There are tools which measure code quality and provide real time code-quality snapshot through reports.
- Great. But reports are not actionable and are rather reactive.
- It makes much more sense if a developer gets to know the error as soon as he makes one.
- Without fixing the error, system doesn't allow him to move further.
- That sounds more proactive, doesn't it?



How to measure Code Quality?

- There are tools which measure code quality and provide real time code-quality snapshot through reports.
- Great. But reports are not actionable and are rather reactive.
- It makes much more sense if a developer gets to know the error as soon as he makes one.
- Without fixing the error, system doesn't allow him to move further.
- That sounds more proactive, doesn't it?



Continuous testing vs. traditional testing

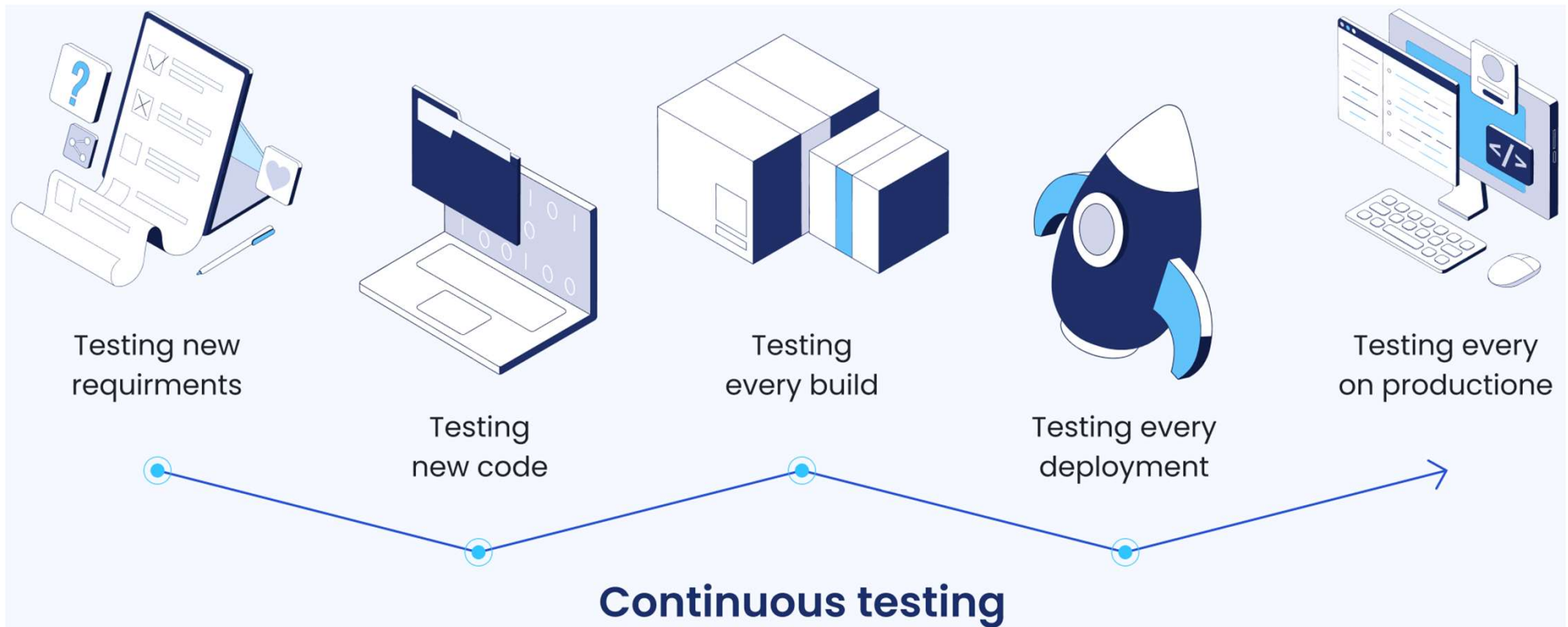
- In traditional software development, testing is often the final stage, preceded by analysis, planning, design, and development.
- This approach is unsuitable for projects with high time-to-market requirements or complex functionality.
- Missing bugs in the early stages of development leads to delayed releases, extended deadlines, and increased development costs.



Continuous testing vs. traditional testing

- Unlike traditional testing, continuous testing does not wait until the specific functionality is finished but checks the code quality as it is written.
- In other words, continuous testing involves verifying code quality in parallel with its creation.
- This allows developers to almost instantly fix bugs when they are discovered and approach the release date with a minimum number of errors and inaccuracies.

Continuous testing vs. traditional testing





Continuous testing vs. traditional testing

Key features of continuous testing include:

- Automated testing, which triggers a check upon the occurrence of an event, for example, pushing a change to the code or creating a build
- Frequent test execution that occurs after each change made
- Seamless integration into the CI/CD pipeline, where development, testing, and deployment automatically replace each other
- Collaboration of testers, developers, and other software development stakeholders
- Shift-left approach, which involves testing at the early stages of the development lifecycle
- Test scaling, which allows increasing or decreasing the number of tests as development progresses



key differences between traditional and continuous testing

Continuous testing

Continuously throughout the software development lifecycle

Timing

Emphasizes test automation

Automation

Frequently, often triggered automatically with each code change or build

Frequency

Provides rapid feedback to developers

Feedback Loop

Integrated with the CI/CD pipeline

Integration

Traditional testing

In dedicated phases after development is complete,

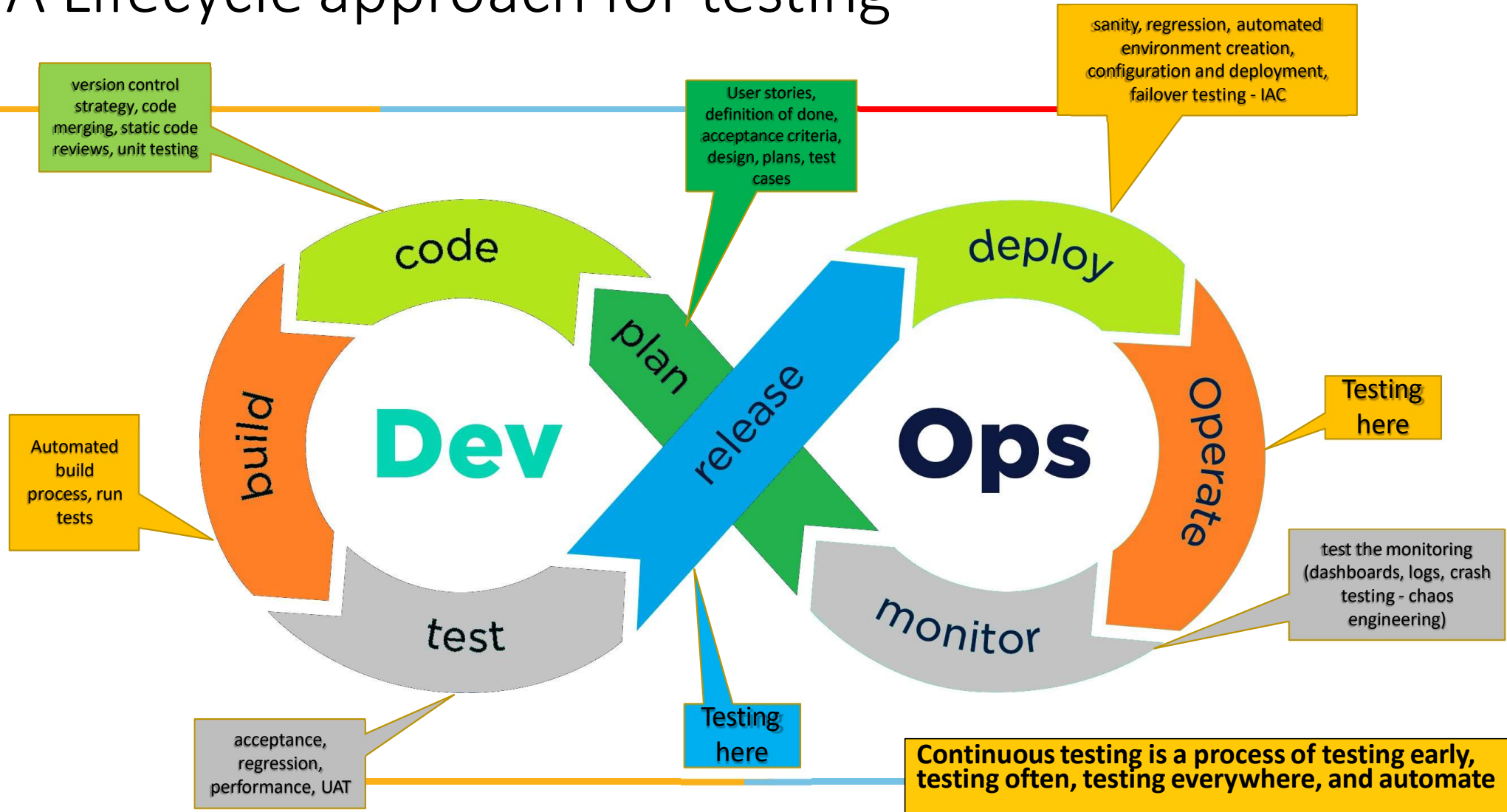
Often relies on manual testing

Less frequently, usually in discrete phases before release.

Provides feedback later in the SDLC,

Often disconnected from the CI/CD pipeline

A Lifecycle approach for testing





Code quality analysis tools- SonarQube

- SonarQube is an open-source platform for code quality inspection.
- It's capable of performing static code analysis to identify bugs, code smells, and security vulnerabilities for a vast variety of programming languages.
- SonarQube supports integration with version control tools like GitHub, Azure DevOps, Bitbucket and GitLab to provide insights for code reviews by performing branch and pull requests analysis.
- Native integration with CI / CD tools like Jenkins enables scheduled or automatic analysis.
- For more information visit the SonarQube official [website](#).



Code quality analysis tools- SonarQube

- The basic steps to add a project into SonarQube and performing your first scan.
- Branch and Pull request analysis.
- Setting up sonar-scanner in Jenkins.
- Automated pull request analysis and pull request decoration.
- Setting up a quality gate check for your GitHub pull requests.
- *Note: Most of the functionalities that will be covered will not be available for SonarQube Community Edition and requires the Developer Edition or higher.*

Code quality analysis tools- SonarQube



- Adding a project into SonarQube
- Adding a project into SonarQube is pretty straight forward.
- Simply click the **Add project** button.



Code quality analysis tools- SonarQube

Analyze your project

We initialized your project on SonarQube, now it's up to you to launch analyses!

- 1 Provide a token**

Generate a token

Generate

The token is used to identify you when an analysis is performed. If it has been compromised, you can revoke it at any point of time in your [user account](#).
- 2 Run analysis on your project**

Enter a name to generate a token

First, you'll need to provide a project key and display name to the project.

- Next, provide a name for the token to be generated.
- This token authenticates the user when it performs an analysis.



Code quality analysis tools- SonarQube

- Once the token is generated, save the token since this is what will be used to perform the scan. You can also use an existing token from another project without having to generate a new one.
- Performing an analysis
- Sonar-scanner is required to perform a scan. Sonar-scanner is available in multiple options including; zip file, docker image, npm package, and a plugin for Jenkins.
- usage: sonar-scanner [options]Options:
 - D,--define <arg> Define property
 - h,--help Display help information
 - v,--version Display version information
 - X,--debug Produce execution debug output



Code quality analysis tools- SonarQube

sonar.host.url=<http://localhost:9000/># must be unique in a given SonarQube instance

sonar.projectKey=Hello-World

sonar.login={myAuthenticationToken}# --- optional properties ---

defaults to project key

sonar.projectName=Hello-World# defaults to 'not provided'

sonar.projectVersion=1.0

Path is relative to the sonar-project.properties file. Defaults to .

sonar.sources=./src

sonar.exclusions=.github/**/* , .vscode/**/* , bin/**/* , docs/**/* , etc/**/* , include/**/* ,
lib/**/* , repository/**/* , share/**/* , .dockerignore , .gitignore , Dockerfile , README.md ,
requirements.txt , sonar-project.properties , assets/**/*

Encoding of the source code. Default is default system encoding

sonar.sourceEncoding=UTF-8

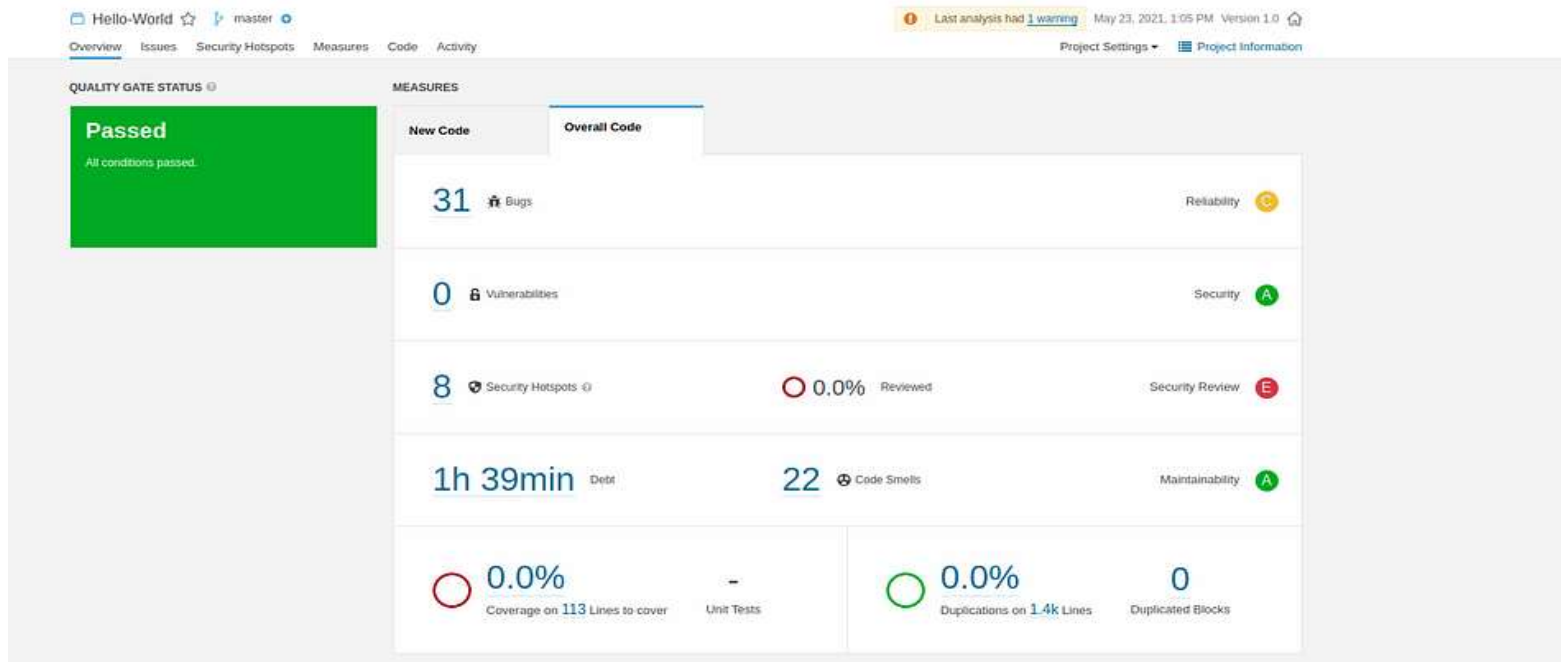
- **Note:** Before an analysis is done the *sonar-project.properties* file needs to be added into the root directory to configure your project. This is a sample *sonar-project.properties* file:



Code quality analysis tools- SonarQube

- The *myAuthenticationToken* is the token given when the project is made in SonarQube.
- Using the zip file
- The sonar scanner zip file can be downloaded [here](#). As of writing, the latest sonar scanner for a Linux-based environment can be downloaded [here](#).
- Once you unzip the file you will see a bin folder. Add the path of the bin folder as a **PATH** variable.
- Navigate to the root directory of the project (make sure the *sonar-project.properties* file is placed) and enter this command in the terminal:
- **sonar-scanner**

Code quality analysis tools- SonarQube



This will start the scan and on completion, provides a URL to view the generated report.
Here is a sample scan report:



Code quality analysis tools- SonarQube

- Alternatively, properties can be passed when running the command with the **-D** option:
- `sonar-scanner -Dsonar.projectKey=Hello-World -Dsonar.sources=. -Dsonar.host.url={SonarQube Host URL} -Dsonar.login={Authentication Token}`
- To run a **branch analysis** (*not available for community edition, requires developer edition or higher*) pass the branch name as a property for sonar-scanner.
- ***sonar.branch.name***: Name of the branch to be scanned.
- Execute the sonar-scanner command with this property or add it to the *sonar-project.properties* file for a branch analysis.
- `sonar-scanner -Dsonar.branch.name=develop`



Code quality analysis tools- SonarQube

- To run a Pull request analysis (*not available in community edition, requires developer edition or higher*) the following properties are required:
- *sonar.pullrequest.key*: Unique identifier of the pull request.
- *sonar.pullrequest.branch*: The name of the branch to be merged.
- *sonar.pullrequest.base*: The branch into which the branch is merged to.



Code quality analysis tools- SonarQube

- Using a Docker Image
- To scan using a Sonar Scanner Docker image, use the following command:

```
• docker run --rm \  
  
-e SONAR_HOST_URL="http://${SONARQUBE_URL}/" \  
-e SONAR_LOGIN="myAuthenticationToken" \  
-v "${YOUR_REPO}":/usr/src \  
--network=host \  
sonarsource/sonar-scanner-cli
```



Code quality analysis tools- SonarQube

- Setting up Jenkins for analysis
- To perform a scan on a codebase using Jenkins, the [Sonar Scanner plugin](#) is required.
- Note, this requires JDK 11 or higher to be installed.
- Once the plugin is installed, configure it to point the scans to your SonarQube instance.
- In Jenkins navigate to **Manage Jenkins** → **Configure System** and add a SonarQube server.
- Press enter or click to view image in full size

- Sample SonarQube configuration for Jenkins

SonarQube servers

Environment variables

☐ Enable injection of SonarQube server configuration as build environment variables

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

SonarQube installations

Name

SonarServer

Server URL

https://[redacted]:9000/

Default is http://localhost:9000

Server authentication token

[redacted] [Add](#)

SonarQube authentication token. Mandatory when anonymous access is disabled.

[Advanced...](#)

[Delete SonarQube](#)



Code quality analysis tools- SonarQube

Automating GitHub PR analysis and PR decoration

Note: *This is not available for the community edition in SonarQube and requires developer edition or higher.*

Steps to be followed:

1. Create a webhook in the GitHub repository.
2. Create a Jenkins job to listen to the webhooks and run the scan.
3. Create and install a GitHub app.
4. Update your SonarQube global settings with your GitHub App information.
5. Add a pull request decoration to the project in SonarQube.



Code quality analysis tools- SonarQube

Creating a webhook

1. In the GitHub repository, navigate to **Settings** → **Hooks**. Click **Add webhook**.

1. Next, provide the following information:

- **Payload URL:** *{Jenkins URL}/generic-webhook-trigger/invoke?token={token}*
(Token is defined at Jenkins Generic Webhook Trigger plugin when creating a job. Install this plugin if required in Jenkins.)
- **Content type:** **application/json**
- Select **let me select individual event** and select **Pull Request** in the check box.



Code quality analysis tools- SonarQube

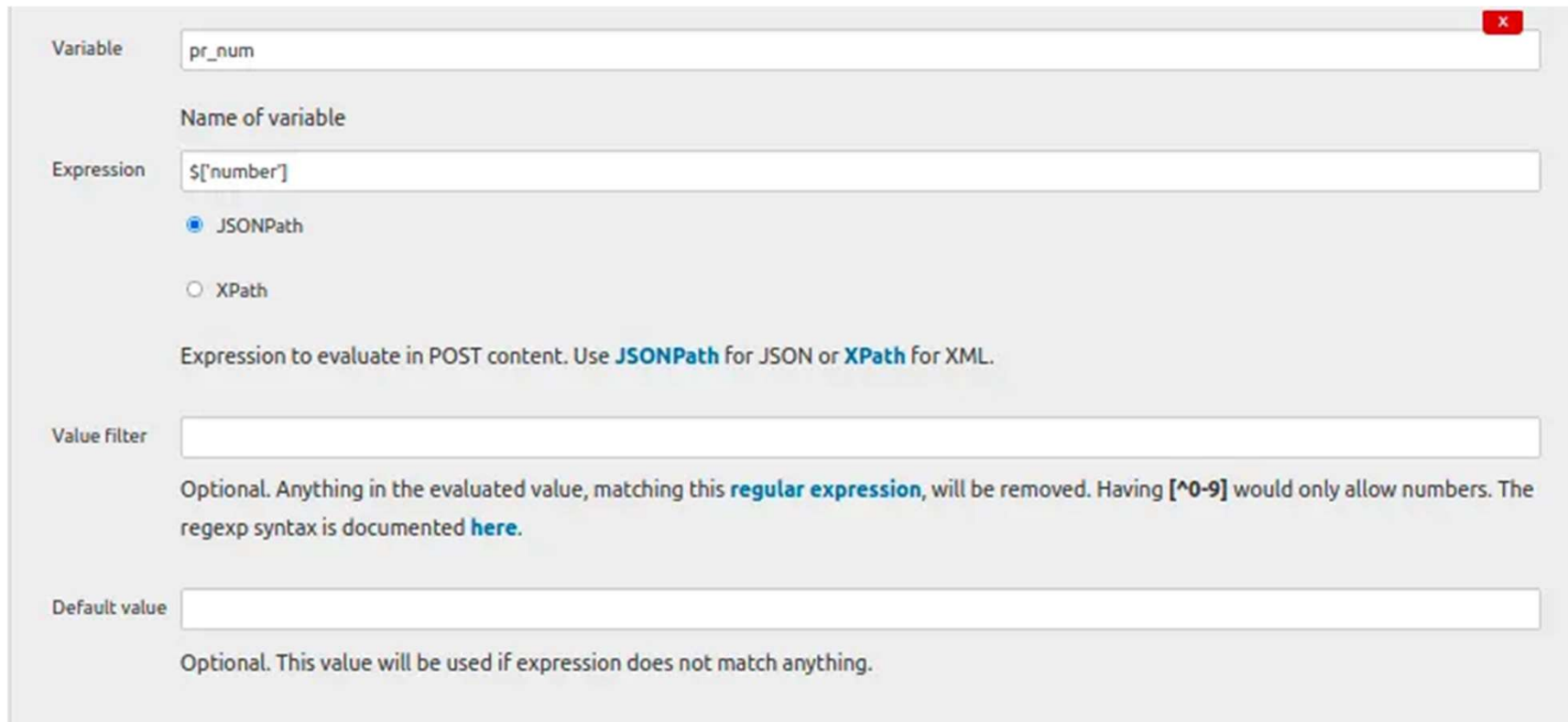
- Create a Jenkins job
- Create a Jenkins job to listen to the webhook triggered by GitHub when a pull request is made and start a SonarQube scan on the branch that has been merged.
- *Step 1. Create a Jenkins pipeline*
- *Step 2. Configure Source Code Management*
- *Repository URL* : URL for the GitHub repository.
- *Credentials*: select a ssh private key. If the key is missing, add the key to Jenkins.
- *Branches to build* : */\${pr_branch}



Code quality analysis tools- SonarQube

- *Step 3. Configure Build Trigger as a Generic Webhook Trigger*
- Add Post content parameters
 - a. pr_num → `['number']` → JSONPath
 - b. pr_base_branch → `['pull_request']['base']['ref']` → JSONPath
 - c. pr_branch → `['pull_request']['head']['ref']` → JSONPath
 - d. action → `['action']` → JSONPath

Code quality analysis tools- SonarQube



Variable

Name of variable

Expression

☒ JSONPath

☐ XPath

Expression to evaluate in POST content. Use **JSONPath** for JSON or **XPath** for XML.

Value filter

Optional. Anything in the evaluated value, matching this **regular expression**, will be removed. Having **[^0-9]** would only allow numbers. The regexp syntax is documented [here](#).

Default value

Optional. This value will be used if expression does not match anything.

Code quality analysis tools- SonarQube

X

Variable

Name of variable

Expression

☒ JSONPath

☐ XPath

Expression to evaluate in POST content. Use **JSONPath** for JSON or **XPath** for XML.

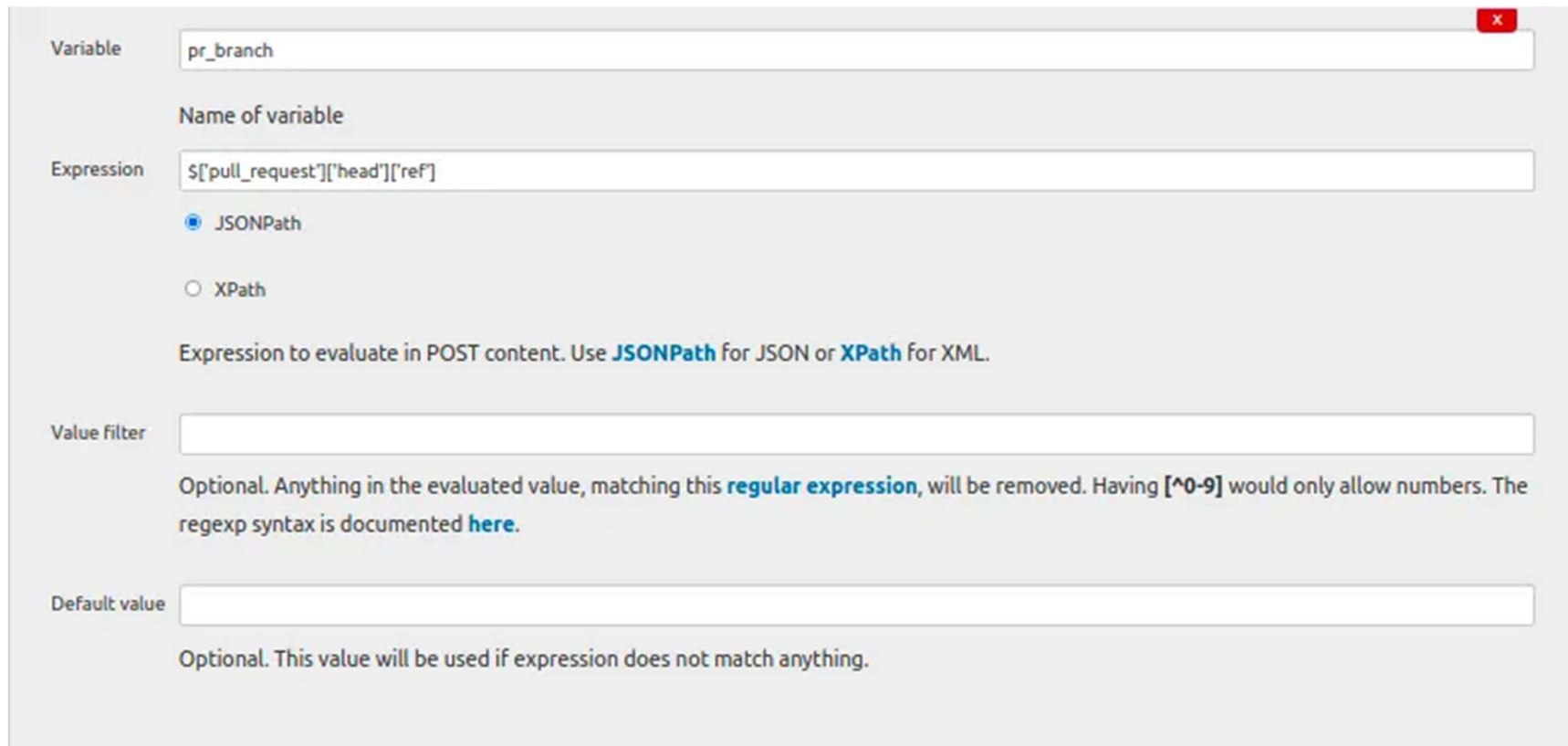
Value filter

Optional. Anything in the evaluated value, matching this **regular expression**, will be removed. Having **[^0-9]** would only allow numbers. The regexp syntax is documented [here](#).

Default value

Optional. This value will be used if expression does not match anything.

Code quality analysis tools- SonarQube



Variable

Name of variable

Expression

☒ JSONPath

☐ XPath

Expression to evaluate in POST content. Use **JSONPath** for JSON or **XPath** for XML.

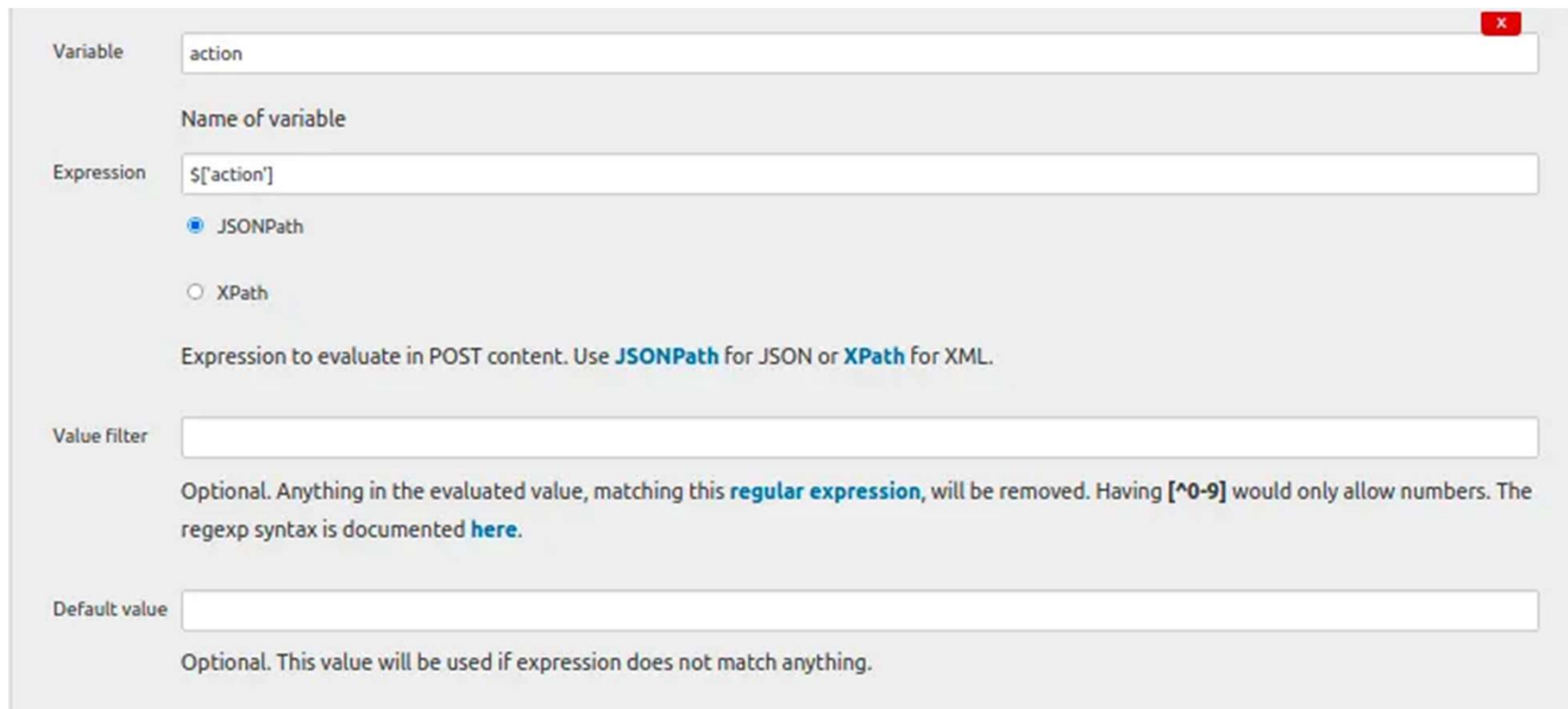
Value Filter

Optional. Anything in the evaluated value, matching this **regular expression**, will be removed. Having **[^0-9]** would only allow numbers. The regexp syntax is documented [here](#).

Default value

Optional. This value will be used if expression does not match anything.

Code quality analysis tools- SonarQube



Variable:

Name of variable

Expression:

☒ JSONPath
☐ XPath

Expression to evaluate in POST content. Use **JSONPath** for JSON or **XPath** for XML.

Value filter:

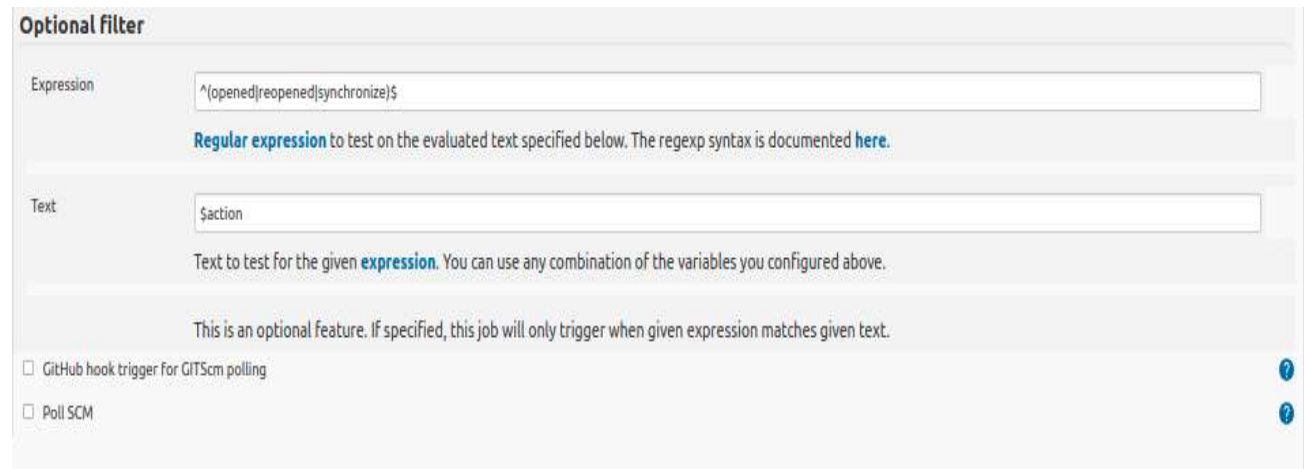
Optional. Anything in the evaluated value, matching this **regular expression**, will be removed. Having **[^0-9]** would only allow numbers. The regexp syntax is documented [here](#).

Default value:

Optional. This value will be used if expression does not match anything.

Code quality analysis tools- SonarQube

- Add **Token** (The token is provided when creating the webhook)
- Add **Cause** to the print parameters (*"Action \$action performed on PR \$pr_num raised for base branch \$pr_base_branch from branch \$pr_branch"*)
- Optional filter
 - a. Text → \$action
 - b. Expression → `^(opened|reopened|synchronize)$`



Optional filter

Expression:
Regular expression to test on the evaluated text specified below. The regexp syntax is documented [here](#).

Text:
Text to test for the given **expression**. You can use any combination of the variables you configured above.

This is an optional feature. If specified, this job will only trigger when given expression matches given text.

☐ GitHub hook trigger for GITScm polling ?

☐ Poll SCM ?



Code quality analysis tools- SonarQube

Step 4. Configure SonarQube Scanner (Build)

- **SonarQube Installation:** Select the installed scanner (which was configured prior).
- **Analysis properties:** copy-paste the sonar-project.properties for **PR analysis** (this includes the 3 properties for a PR analysis mentioned above).
- Save the job.
- Now, when a change is made and the PR is created, the hook should be fired from GitHub and the job should be triggered automatically.



Code quality analysis tools- SonarQube

- Create a GitHub app
- Refer to GitHub's documentation on [creating a GitHub App](#) for general information on creating your app.
- Specify the following settings in your app:
- GitHub App Name: Your app's name.
- Homepage URL: You can use any URL, such as <https://www.sonarqube.org/>.
- User authorization callback URL: Your instance's base URL.
- For example, *https://yourinstance.sonarqube.com*.



Code quality analysis tools- SonarQube

- Webhook URL: Your instance's base URL. For example, <https://yourinstance.sonarqube.com>.
- Grant access to the following Repository permissions:
 - ★ Checks → *Read & Write*
 - ★ GitHub Enterprise: Repository metadata / GitHub.com: Metadata → *Read & Write*
 - ★ Pull Requests → *Read & Write*
 - ★ Commit statuses → *Read-only*
- If you are setting up **GitHub Authentication**, in addition to the mentioned repository permissions, grant access to the following:



Code quality analysis tools- SonarQube

- User permission: *Email Address* → *Read-Only*,

Organization permissions: *Members* and *Projects* → *Read-only*.

- Under “Where can this GitHub App be installed?,” select **Any account**.
- Next, the created app needs to be installed.
- Refer to the GitHub documentation for [installing GitHub apps](#).



Code quality analysis tools- SonarQube

- Update your SonarQube global settings with your GitHub App information
- After the GitHub app is installed, the next step is to update the SonarQube settings to allow for importing of GitHub projects.
- In your SonarQube instance navigate to **Administration Configuration → General Settings → ALM Integrations GitHub** and specify the following settings:
- **Configuration Name** (Enterprise and Data Center Edition only):
 - The name is used to identify your GitHub configuration at the project level.



Code quality analysis tools- SonarQube

- **GitHub URL:** For example, *<https://github.company.com/api/v3>* for GitHub Enterprise or *<https://api.github.com/>* for <http://GitHub.com>.
- **GitHub App ID:** The App ID is found on your GitHub App's page on GitHub at Settings Developer Settings GitHub Apps.
- **Client ID:** The Client ID is found on your GitHub App's page.
- **Client secret:** The Client secret is found on your GitHub App's page.
- **Private Key:** Your GitHub App's private key. You can generate a .pem file from your GitHub App's page under Private keys. Copy and paste the contents of the file here.



Code quality analysis tools- SonarQube

- Add pull request decoration to the project in SonarQube
- Finally, add the following setting to your project in SonarQube to enable pull request decoration.
- Navigate to Project Settings → General Settings → Pull Request Decoration and add:
- Configuration name: The name of the configuration that was added prior.



Code quality analysis tools- SonarQube

- **Repository identifier:** The path of your repository URL.
- Enable “Analysis Summary” under the GitHub Conversation tab.
- This enables you to view a summary of the analysis in the GitHub conversation tab.
- All done.
- Now when a pull request is opened automatically, Jenkins would start a SonarQube analysis and return the summary of the report to GitHub.



Code quality analysis tools- SonarQube

- Setting up Quality Gate check for Pull requests
- To ensure that all pull requests are blocked from merging if the branch fails to pass the SonarQube quality gates, add the following setting into the repository:



Code quality analysis tools- SonarQube

- Code quality is often deprioritized by developers even though it is essential.
- Benefits of quality code are numerous, but to name a few, it increases readability by other developers and it's easily shared across teams, fewer tech debts, and higher sustainability.
- Poor code dies early.
- A code that cannot be read and understood by another developer cannot be easily edited, therefore it will be replaced within a short period of time.
- This is why quality is very important for a software's life cycle.
- “Quality means doing it right when no one is looking.” - Henry Ford



QUESTIONS AND DISCUSSION THANK YOU