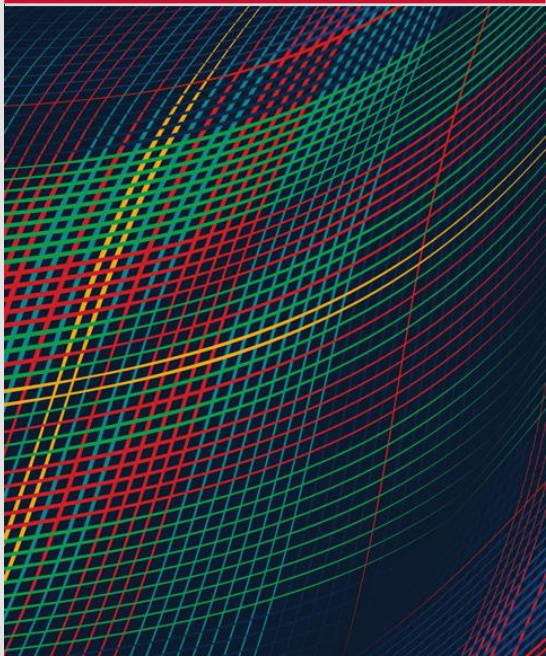
Carnegie Mellon University
Software Engineering Institute

Navy Software Testing Strategy

MARCH 1, 2025

Alexander Volynkin, Ph.D.
Brent Clausner
Joseph Yankel



Document Markings

DM25-0185

Carnegie Mellon University
Software Engineering Institute

Presentation Name

Introduction and Roadmap

What This Brief Covers

This presentation provides practical strategies for reducing technical debt and improving maintainability in legacy systems through test automation. It also includes general strategy that would apply to new software system acquisition.

Case Studies - Examples of successful test automation adoption

Testing Strategies - Approaches in testing legacy, new, and cyber physical systems

Automated Testing - Best practices for effective test automation

Hardware In The Loop - Strategies for testing cyber-physical systems

Configuration Management - Best practices in CM, considerations for separate test organizations

Roles and Responsibilities - List of various roles related to testing and their functions

Software Tools - Lists of various tools and their usage

The notes area of this briefing is used to articulate additional information.

Framing the Problem

Legacy systems suffer from high maintenance costs, brittle codebases, and slow development cycles.

Solution: Test automation is a key enabler for reducing technical debt and improving maintainability.

Approach: Introduce positive patterns to adopt and anti-patters to avoid.

Key Positive Patterns

- Incremental automation starting with high-value, stable components
- Prioritizing regression testing for critical workflows
- Leverage hardware-in-the-loop testing when applicable
  - HIL testing is time consuming, and requires considerable planning
- Gradually refactor, start small and improve testability over time
- Utilize Configuration Management best practices

Key Anti-Patterns

- Replacing all manual tests at once with automation efforts
- High-reliance on brittle UI testing
- Ignoring necessary infrastructure improvements or not demanding Infrastructure As Code (IaC) and Configuration as Code (CaC) utilization
- Lack of clear ownership for identifying test maintainers

General Testing Facts

Large (Slow)

Typical Large End-to-end UI

Medium

Typical Medium External Services Single UI

Small (Fast)

Typical Small Individual Classes

Have as many cheap, fast running tests as possible and minimize the number of expensive and slow tests.

DoD Acquisition Requirements Are Unique

DoD requires large systems to plan for and undergo independent Operational Test & Evaluation

- Planned and executed by a different organization than the Program Office acquiring the software system
- Requires an early Test and Evaluation Strategy more compatible with a “big bang” delivery than the incremental delivery typical in agile

USAF guidance (AF99-103) now aligns independent testing with a more incremental approach

- Integrated testing and integrated test teams are a specific strategy that is highlighted
- Incremental testing is specifically discussed and encouraged prior to full operational testing of a deployed capability

Test strategy needed that accounts for software not being done, requiring maintenance.

Presentation Name

Case Studies

Carnegie Mellon University
Software Engineering Institute

Case Study 1: Testing Software/Hardware Contractor Deliverables for Navy Cyber-Physical System

Carnegie Mellon University Software Engineering Institute

Ship hardware controlled by HMI/PLC with extensive software and firmware functionality that is composed of a mix of ladder logic and C/C++ firmware code

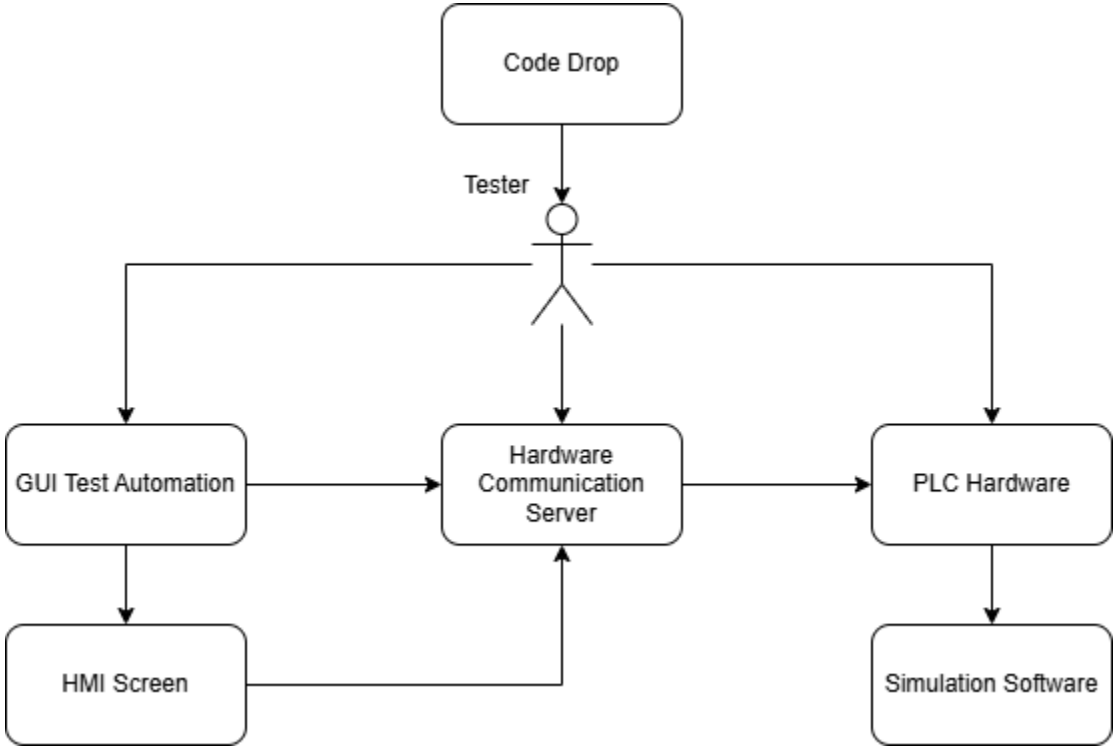
Initial State:

- Multiple hardware units. Each must be developed and tested individually
- Software and portions of hardware developed by contractor
- Software has significant GUI component. User Interface displayed on specialized hardware
- NAVY must do integration and acceptance testing of each component. Many components are identical and must be tested individually
- Manual process throughout: test deployment, test configuration, testing and log analysis

Initial State of Testing

- GUI testing is automated with Robot-based test automation tool
- Communication with real hardware maintained using available vendor tools
- Code is loaded manually onto hardware
- Test environments are set up using a slow, manual process (may take days)
- Complex test environment with many hardware components.
- Lack of simulation
- Test reports are stored and managed manually
- Manual correlation between code drops and tests performed (prone to human error)

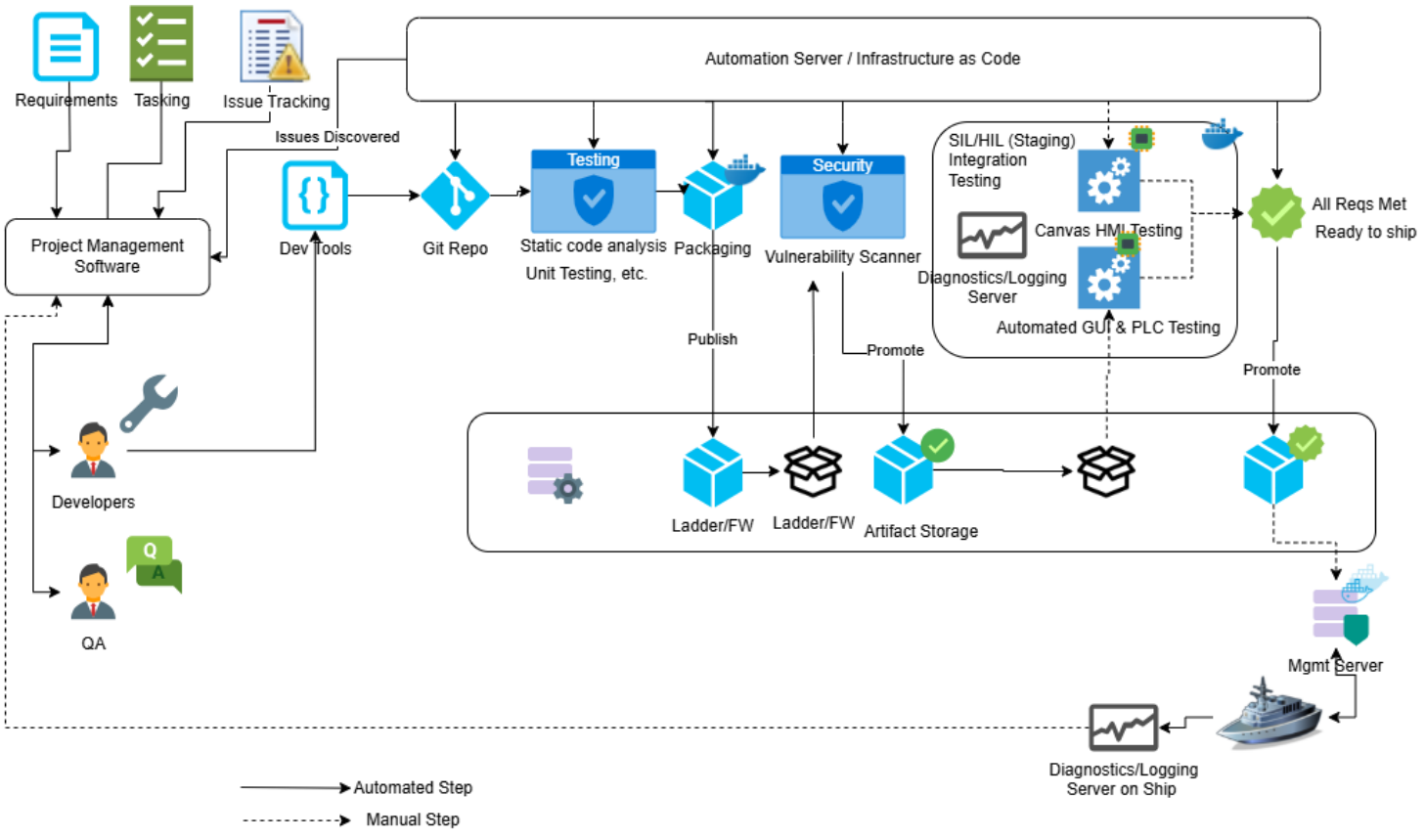
Approach: Build Initial Test Workflow



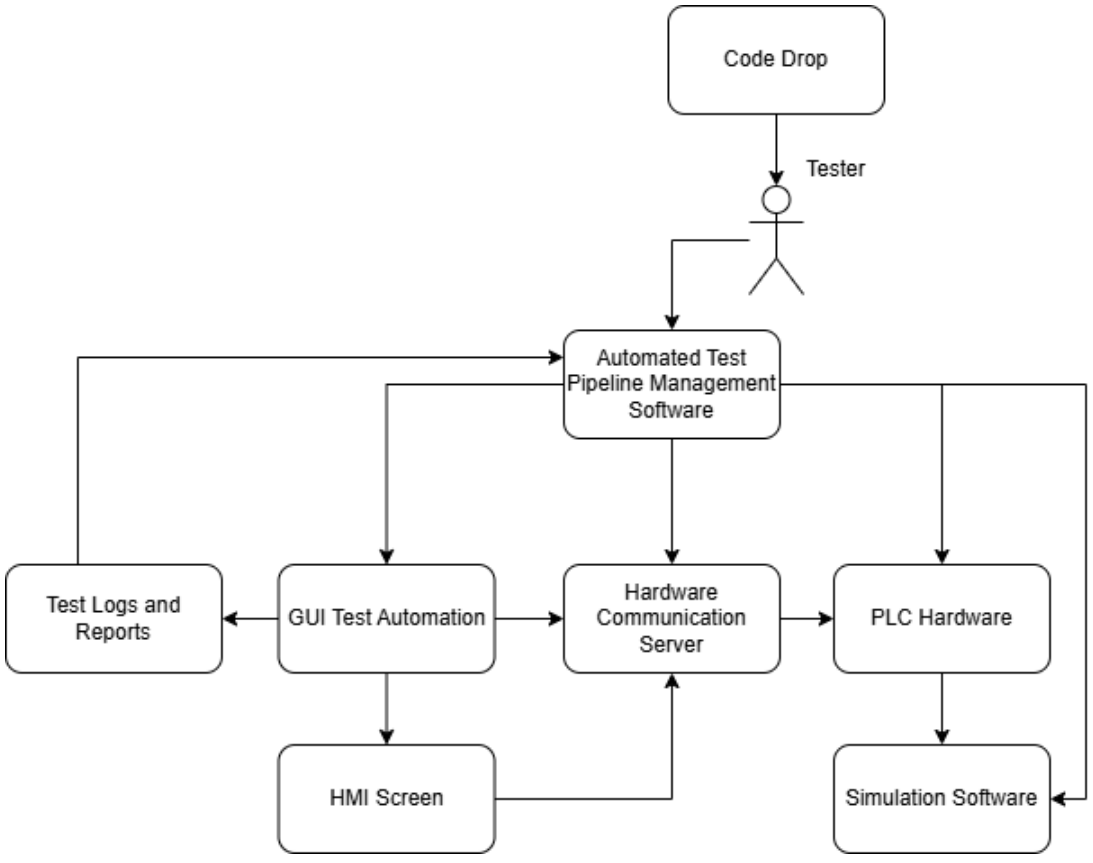
Introduce Automation Test Pipeline

- Offline or cloud operation
- Gitlab for CI/CD and version control
- Repeatable clean state for each deployment
- Test deployment on code commits (Integrate with CI/CD)
- Automated hardware provisioning
- Automatic hardware communication server configuration
- Simulation kits configuration and deployment
- Background static analysis on Ladder Logic code
- Test Log aggregation storage, reporting and dashboarding

Automated Test Pipeline Architecture



Current State Test Workflow



Static Code Analysis
Security and Stability

Test Log Dashboard
Statistics overtime, Long term storage

Consistent Test Environment
Stable Repeatable results

Automated provisioning
All components of pipeline

Hardware Simulation

Test Artifacts and Deployment Package
Generation and Storage

Case Study 2: Testing on multiple versions of Windows OS (8, 10, 11)

Carnegie Mellon University Software Engineering Institute

Software is obtained that is capable of being deployed on different versions of windows

Requirements:

- Need to ensure no errors or any faults are encountered when executing the software
- Memory usage needs to not grow significantly
- CPU usage also needs to be monitored that doesn't cause additional cycles
- No additional network traffic is used and what is used is logged and verified
- All output is captured and logged
- Storage of the software is necessary for longevity
- Software usage must be well-documented for users to be able to quickly use it.
- Eliminate slow, error prone test environment creation

Approach used to automate testing

- Utilize a container or virtual machine to host the OS
- Use a shared mount between host and container/VM to store files from test
- Use screenshots as needed to capture system responses or state
- Define and verify pass/fail criteria
- Cleanup
  - Container/VM is shutdown at end of test execution
  - Container/VM state needs restored to original, clean state
- Provide a means to connect to a shell within the container/VM
- Keyboard/mouse simulation may be required to interact with application under test
- Use CI/CD pipelines to automate testing, infrastructure creation, and delivery of software artifacts

Lessons Learned

- Cyber physical systems are much more complicated to test than software alone
- Utilize automation for delivery of artifacts into various staging environments for further testing
- Require simulation software for both hardware and software components as contractual agreements.
- Establish hardware-in-the-loop testing as early as possible, substituting simulation/emulation early while waiting on hardware
- Don’t reinvent the wheel. Use well known techniques, existing test automation frameworks and processes
- Automation may require a significant amount infrastructure resources, so it is important to discover early

Carnegie Mellon University
Software Engineering Institute

Presentation Name

Testing Strategies for Legacy Systems

Challenges in Testing Legacy Software

- Lack of existing automated tests
- Poor modularity and testability
- Unclear or outdated requirements

Effective approaches

- Use strangler pattern for testing. Instead of a complete rewrite, begin by adding new features around the edge of the existing system and gradually replacing the old with the new.
- Use a proxy or façade layer to act as an intermediary, to test either the legacy system or direct to the new functionality being updated.
- Ensure Unit, Integration, and Regression tests are updated and tested as new features are developed.
- Execute smoke tests first and automate critical-path tests to ensure basic system stability

Automated regression tests that focus on key features helps to ensure expectations.

Best Practices

- Testing is necessary when new functionality or data is added, and whenever deployment configuration or infrastructure is changed.
- Priority should be given to compatibility, performance, and regression testing, but limited to new functionality and configuration.
- Compatibility testing ensures that new features are compatible with the existing system. Check for conflicts or dependencies that might arise from integrating new features with legacy code.
- Performance testing ensures that the new features do not degrade the performance of the legacy system.

Modular software lends itself better to application of the strangler pattern more easily..

Carnegie Mellon University
Software Engineering Institute

Presentation Name

Testing Strategies for New Systems

Modern Strategies May Be Adopted Over Time

- Shift-left testing requires early unit and integration tests
- Continuous Integration (CI) is achieved by using automated test suites and executing tests with every build
- Feature toggles, A/B testing, and canary releases are approaches that allow faster and safer deployments
- Code review isn't just for application code, test code should be reviewed also

Best Practices

- Similar approaches can be used for software and HIL testing
- Organize and introduce test procedures as early as possible in the development cycle
- If software is being developed by a contractor, consider specifying test engagements in the contract. Early integration testing is crucial for quality and stability of the software. Contractors deliver test harnesses, including simulator/emulator configurations with the main software deliverable
- Ensure compatibility of automated testing procedures across all software components subject to integration

Carnegie Mellon University
Software Engineering Institute

Presentation Name

Testing Strategies for Cyber Physical Systems

Testing with Hardware in the Loop [1/2]

- Integration testing with Hardware early is important as it helps eliminate human errors
- Consider adding automated tests against Hardware
- Use consistent and repeatable test environment/configuration
- Regular software testing pipeline may be sufficient with additional hardware testing harnesses. Examples of such harnesses include:
  - Hardware provisioning – automated loading of firmware into hardware components
  - Hardware/Software simulation

Testing with Hardware in the Loop [2/2]

- Simulate Hardware when you can for unit, sub and system tests
- Don’t expect for simulation to completely replace Hardware CI
- Don’t wait until end-to-end testing to test with real Hardware components
- Perform Hardware “arming” tests frequently
- HW/SW configuration for reliable CI is very challenging unless simulating
- Consider full memory snapshots for SW and HW components

Key Strategies for Introducing Regression Tests [1/2]

- Incrementally add tests, beginning with the most critical functionality before expanding
- Prioritize tests that exercise the most frequently used functionality and most likely impacted by changes
- Perform a risk assessment to identify highest error rate areas of the application. This can be done by reviewing issues creating during development that are tagged ‘bug’. NOTE: requires a test strategy that mandates good issue tracking and tagging
- Use the same frameworks in place for existing tests and continue to add to them

Key Strategies for Introducing Regression Tests [2/2]

- Collaborate with sustainment teams to review tests and verify efficacy
- Automate all new functionality and add to regression test suite
- Regularly review and maintain tests
- As part of a test strategy, ensure test maintenance is an area that is covered and financially supported

Version Controlled Tests [1/2]

- Tests should be version controlled to be specific to version of software
- Newer versions of software can have new tests added
- New version of software can have older tests decommissioned
- Features can change with new versions, so tests may need updated to utilize features
  - Features can be removed
  - Modules can be deprecated
- These tests are regression tests
- Allows for faster turnaround for testing patches

Maintain a branching structure like the version of software. Major, minor, and patch.

Version Controlled Tests [2/2]

- Software can have multiple releases
  - Example: 12.0.4: (<major_version>.<minor_version>.<patch>)
- Any number of updates are possible to occur at different versions
  - Most Major versions
    - Some features modified/removed/added
  - Minor versions
    - Some features have some underlying functionality modified
    - Most features stay but can be deprecated
  - Patches
    - Possibly targeted to fix security issues
    - Contain fixes for specific issues

Avoid test automation gotchas

- Much more work converting manual to automated testing than expected
- Don’t automate tests for unstable interfaces
- Do use exploratory testing
- Do automate tests for APIs, stable user interfaces
- Tools don’t automate test design and judgement
- Capture/replay usually results in breakage and/or test script re-recording
- Set expectations for test asset maintenance
- Automate performance testing incrementally
- Follow test automation design patterns

Consider carefully what should not be automated and automate everything else.

Presentation Name

Best Practices in Automated Testing

Carnegie Mellon University
Software Engineering Institute

Best Practices In Automated Testing

- Automated tests take a manual workflow, make it have scripts or other software execute a series of steps and verify that output matches expectations
- Testing that takes a large amount of time can benefit from automation
- Smaller tests can be run quickly and possibly in parallel
- Prioritize automated tests that can cover most modules and take less time to run/write
- Utilize CI/CD Pipelines to run automation on code commits and/or merges

Prioritize writing automated tests that cover most modules and are quick to write.

Functional Testing Automation

Carnegie Mellon University Software Engineering Institute

Focus on automating existing manual integration tests and prioritize:

- Manual tests that cover high priority requirements
- Existing integration tests that cover the most functionality

Unit-testing may be faster to automate but functional coverage is more important.

New integration tests should then be automated after completing automation of the existing tests.

Automated Software Testing Types [1/3]

Unit testing

- Consists of testing functions and methods of an application.
- Easy to write and quick to execute
- Requires source code
- Easily added to the build system

Integration testing

- Tests functionality between different modules
- Automated tests rely on modules being either complete or near complete

Automated Software Testing Types [2/3]

System testing

- Verifies that end-to-end system testing meets all specified requirements
- Automated tests can take longer to write and run

Acceptance testing

- Verifies that specific requirements are satisfied
- Possibly includes performance-based tests
- Faster to write than system tests

Automated Software Testing Types [3/3]

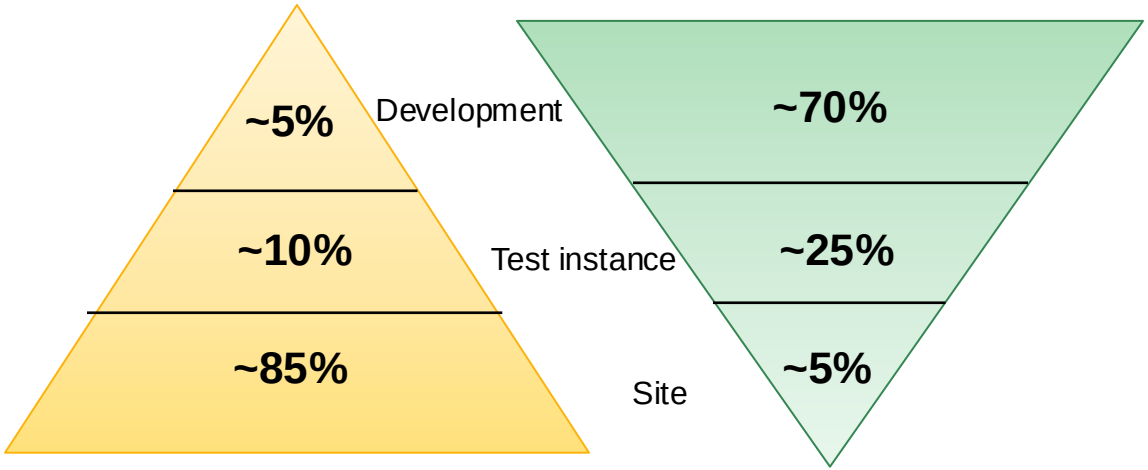
Regression testing

- Tests previously executed are re-executed after a release
- Includes any of the previous testing types
- Must be maintained and updated as new functionality is added

Smoke Testing

- Smoke testing focuses on speed - a quick, less comprehensive regression test
- Typically executed as a predetermined set of tests on a build of software after any significant change.
- Excellent for quickly catching issues on tests that may normally take a long time to execute.
- Example: Instead of executing 1000 tests, just execute a few tests by category

Goals for Agile Process with Automated Testing



As Is

Should be

Increase team, DT, and OT confidence in code quality

- Improve communication with DT/OT what kind of testing has passed
- Must test more of the codebase sooner and faster

Test Manager, Major Defense System, 14 Dec 2016

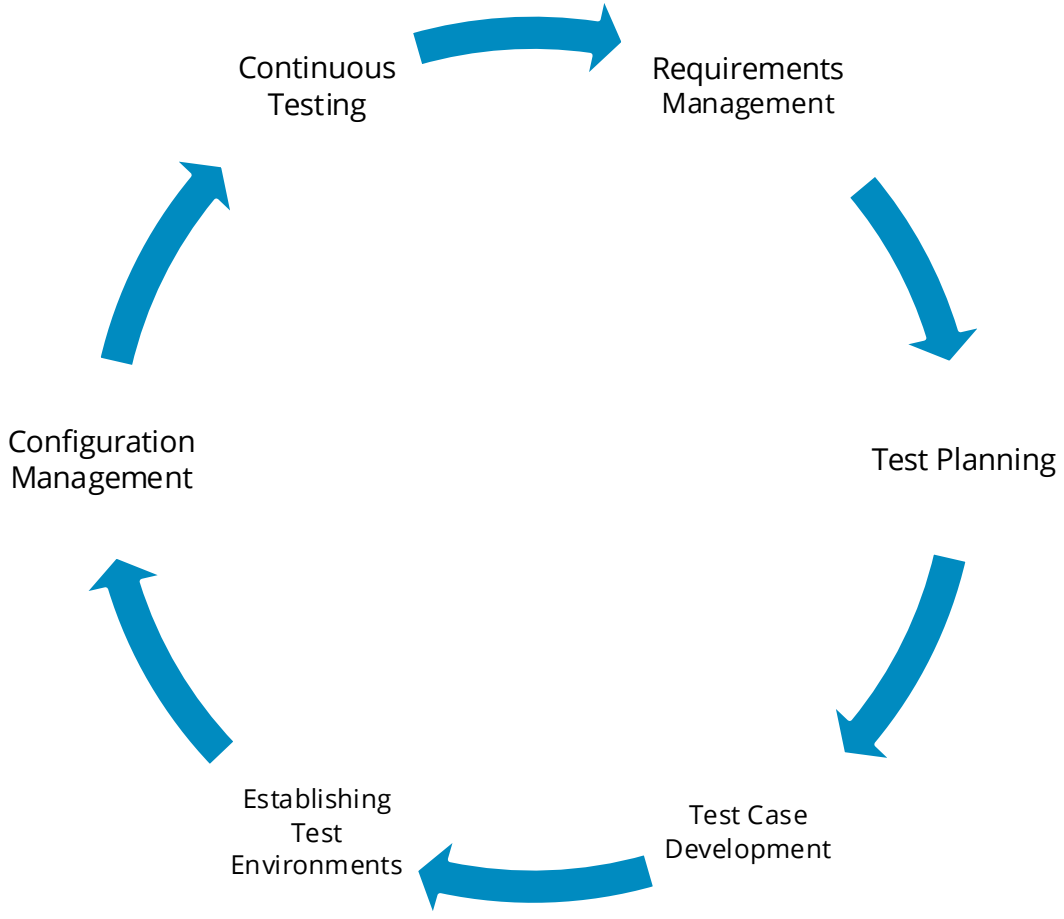
More testing is done on delivery than at time of development.

Presentation Name

Test Life Cycle

Carnegie Mellon University
Software Engineering Institute

Test Life Cycle



Requirements Management: Identification

Carnegie Mellon University Software Engineering Institute

Requirement Identification - Gather and document all functional and non-functional requirements from stakeholders.

- **Clarity and Focus**: Help the team understand what the software is supposed to do
- **Prioritization**: Helps prioritize which features and aspects of the software are most critical to test
- **Test Coverage**: Ensuring that all requirements are identified makes it possible to create test cases that cover all scenarios, reducing the risk of missing critical bugs
- **Traceability**: Helps trace test cases back to requirements, making it easier to understand the scope of testing and maintain accountability
- **Early Detection of Issues**: Helps detect issues early in the development process, reducing costly fixes later

Requirements Management: Prioritization

Requirement Prioritization - Rank requirements based on their importance and impact on the project.

- **Resource Allocation**: Projects often have limited time and resources. Prioritizing requirements ensures that the most important features are tested first, making optimal use of available resources
- **Risk Management**: By identifying and focusing on high-priority requirements, testers can address areas with the highest risk first, catching critical defects early in the process
- **Clarifies Goals**: It brings clarity to development and testing goals, aligning the team to work towards the most important objectives and avoid unnecessary workload
- **Incremental Delivery**: In agile and iterative development models, prioritization allows for incremental delivery of features, ensuring that each iteration includes the most critical functionality

Requirements Management: Traceability Matrix

Traceability Matrix - Create a traceability matrix to map requirements to corresponding test cases.

- **Traceability**: It provides a way to trace the requirements through the development lifecycle, ensuring that each requirement is addressed by design, implementation, and testing efforts
- **Requirement Coverage**: It ensures that all requirements are covered by test cases, which helps in identifying any gaps in testing
- **Impact Analysis**: When changes occur, a traceability matrix helps in assessing the impact on the various aspects of the project, making it easier to manage changes
- **Accountability**: It defines a clear path of responsibilities, indicating which team or individual is responsible for each requirement and its associated test cases
- **Compliance**: For projects requiring adherence to standards, a traceability matrix ensures that all standards and legal requirements are met

Requirements Management: Version Control

Carnegie Mellon University Software Engineering Institute

Version Control - Manage changes to requirements through version control to ensure consistency (Jira, Azure, etc.).

- **Historical Tracking**: Version control allows you to track the history of changes made to requirements. This is critical for understanding how and why requirements have evolved over time
- **Baseline Management**: Helps in managing different baselines, such as initial requirements, agreed-upon changes, and final developed requirements
- **Compliance and Auditing**: For projects that require compliance with regulations or standards, version control provides an audit trail that shows how requirements were managed and implemented

Requirements Management: Review and Validation

Carnegie Mellon University Software Engineering Institute

Regularly review requirements to identify and resolve ambiguities, inconsistencies, and gaps.

- **Ensuring Accuracy**: Helps verify that requirements accurately reflect the needs and expectations of stakeholders. It reduces the risk of misinterpretations or misunderstandings
- **Improving Testability**: Clear, precise, and well-defined requirements are easier to translate into test cases, ensuring comprehensive and effective testing
- **Early Detection of Issues**: Early reviews and validation can uncover potential problems or conflicts before they become costly to fix later in the development process

Test Planning [1/2]

- **Define Scope and Objectives** - Clearly outline what needs to be tested and the goals of the testing process
- **Identify Test Types** - Determine the types of tests to be conducted, such as unit, integration, system, and acceptance tests
- **Resource Allocation** - Assign roles and responsibilities to the testing team, ensuring adequate resources are available
- **Create Test Environment** - Set up the necessary hardware, software, and network configurations to replicate the production environment
- **Develop Test Schedule** - Establish timelines for test preparation, execution, and review, along with milestones and deadlines

Test Planning [2/2]

- **Risk Management** - Identify potential risks and create mitigation strategies to address them
- **Test Data Preparation** - Generate or acquire test data that accurately reflects real-world scenarios
- **Test Case Design** - Develop detailed test cases that cover all functional and non-functional requirements
- **Review and Approval** - Conduct reviews of the test plan with stakeholders and obtain necessary approvals
- **Monitor and Adjust** - Continuously monitor the testing process and adjust as needed to stay on track and achieve objectives

Test Case Development [1/2]

Carnegie Mellon University Software Engineering Institute

Understand Requirements

- Thoroughly review the software requirements, specifications, and acceptance criteria
- Identify the scope and objectives of the testing

Identify Test Scenarios

- Based on the requirements, list all possible test scenarios
- Consider both positive (expected to pass) and negative (expected to fail) scenarios

Define Test Case Structure

- Establish a clear structure for test cases, typically including:
  - Test Case ID: Unique identifier
  - Test Case Description: Brief description of the test case
  - Preconditions: Any setup required before the test
  - Test Steps: Step-by-step instructions to execute the test

Test Case Development [2/2]

Carnegie Mellon University Software Engineering Institute

Define Test Case Structure (cont.)

- Test Case ID: Unique identifier
- Expected Results: The expected outcome of each step
- Post-conditions: Any cleanup required after the test

Prepare Test Data

- Identify and prepare the necessary test data required for the test cases
- Ensure the test data is representative of real-world scenarios

Add Preconditions

- Document any prerequisites or setup needed before executing the test case
- This includes the initial state of the system, configurations, or specific data setup

Establishing Test Environments [1/3]

- Install automated testing tools (e.g., Selenium, PlayWright)
- Set up performance testing tools (e.g., LoadRunner, JMeter)
- Generate or obtain test data that accurately represents real-world scenarios
- Configure Version Control Systems:
  - Set up repositories (e.g., Git) to manage code and track changes
  - Implement branching strategies for different testing phases

Establishing Test Environments [2/3]

Establish Continuous Integration/Continuous Deployment (CI/CD):

- Configure CI/CD pipelines (e.g., Jenkins, GitLab CI) for automated testing and deployment
- Integrate testing into the CI/CD pipeline for regular and automated test runs

Define Test Cases and Scripts:

- Write and organize test cases for various scenarios
- Develop automated test scripts if applicable

Establishing Test Environments [3/3]

Carnegie Mellon University Software Engineering Institute

Set Up Monitoring and Logging:

- Implement monitoring tools (e.g., New Relic, Splunk) to track system performance and issues
- Set up logging mechanisms to capture and analyze test results

Perform Test Environment Validation:

- Verify the environment setup by running sample test cases
- Ensure that the environment replicates the production setup closely

Document and Communicate:

- Document the setup process, configurations, and any issues encountered
- Share the documentation with the testing team and stakeholders

Monitoring environments and communication help to keep stable environments.

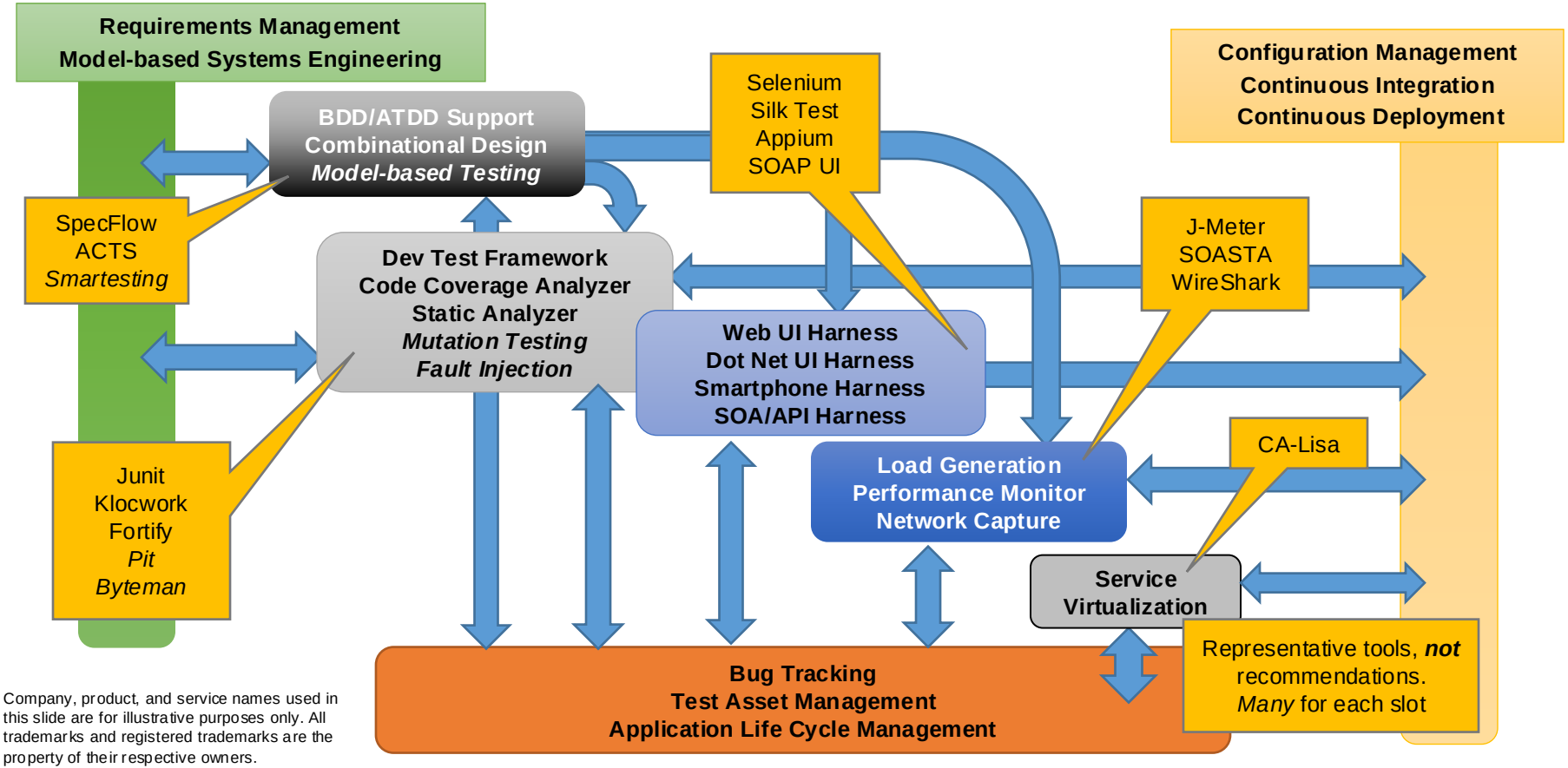
Test Execution [1/2]

- Testing as early as possible, considered shifting the testing left
- Automated tests should have no intermittent failures
- Non-performance based automated tests should be added into the CI/CD Pipeline
- Unit tests should be included into the build
  - Verify as the system is compiled.
- Integration testing
  - Executed after unit-testing
  - Development should be complete or near complete

Test Execution [2/2]

- System testing executed after integration testing
- Acceptance testing executed after system tests
- Regression testing
  - A new release of software should have the acceptance tests re-run
  - All automated tests that are capable of being run, should be run
  - Pipeline can be setup to test new software upon deployment with regression tests
- Test environment needs to be stable for executing tests at any stage

Test Automation Reference Architecture



Continuous Testing [1/2]

- **Define a Testing Strategy**: Clearly outline the scope, objectives, and types of testing (e.g., unit, integration, system, acceptance). Ensure the strategy aligns with goals

- **Test Automation**: Automate as many tests as possible. Begin with unit tests, then move on to integration tests, system tests, and user acceptance tests. Tools like Selenium, JUnit, and TestNG can be helpful

- **Continuous Integration (CI)**: Integrate code changes frequently and automatically run tests upon each integration. This helps detect issues early. Use CI tools like Jenkins, Gitlab, etc.

- **Continuous Deployment (CD)**: Automate the deployment of software to various environments (development, testing, staging, production). This ensures that every change that passes the tests is automatically deployed to the next stage

Continuous Testing [2/2]

- **Environment Management**: Use containerization tools like Docker to ensure consistency across different environments. This helps in replicating production-like environments for testing

- **Test Data Management**: Strategy for managing test data. This includes generating, masking, and maintaining test data that mimics real-world scenarios

- **Monitoring and Feedback**: Implement monitoring tools to track the performance and stability of the application in real-time. Use feedback from these tools to improve the testing process continually

Test Life Cycle summary

- Planning tests, focus on requirements.
  - Planning can expose testing environmental needs
- Testing environments must be stable
- Executing tests should be done early and automated with CI
- Tests should be stored and version controlled
  - Testcases complete with steps and description are stored in a test case manager
  - Automation should be version controlled (e.g., IaC scripts)

Software Test Tools

- Jira add on for configuration management

- Xray test management for jira
  - Allows test case creation
  - Test Plans
  - Ties into EPICs for requirement linking

Carnegie Mellon University
Software Engineering Institute

Presentation Name

Configuration Management

Considerations when test organizations are separate from the development organizations as seen in DT/OT scenarios in government or when dealing with sustainment/maintenance.

Best Practices [1/2]

- **Define Configuration Items**: Identify the items that need to be managed, such as source code, test scripts, test data, documentation, and any dependencies
- **Establish Version Control**: Use version control systems (VCS) like Git or Bitbucket to manage changes to configuration items. Ensure that all changes are tracked and maintain a history of revisions
- **Set Up Build Automation**: Implement build automation tools like Jenkins, Maven, or Gradle to automate the process of compiling, assembling, and testing the software. This ensures that builds are consistent and reproducible
- **Manage Environments**: Clearly define and document the configurations for different environments (development, testing, staging, production). Use tools like Docker or Vagrant to create consistent and isolated environments

Best Practices [2/2]

- **Implement Continuous Integration (CI)**: Integrate code changes frequently and automatically run tests using CI tools. This helps to detect and fix issues early in the development process.
- **Automate Deployments**: Use deployment automation tools to deploy software to different environments. This reduces manual errors and ensures that deployments are consistent.
- **Configuration Auditing**: Regularly audit configurations to ensure compliance with policies and standards. This helps in maintaining the integrity and security of the software.
- **Issue Tracking and Change Management**: Use issue tracking systems like Jira or Bugzilla to manage bugs, enhancements, and tasks. Implement a change management process to review and approve changes before they are applied.
- **Backup and Recovery**: Implement a backup and recovery plan for critical configuration data. Ensure that you can restore configurations in case of a failure or data loss.

Establish Clear Ownership and Boundaries

Define responsibilities for each organization and document them in a formal agreement (e.g., RACI matrix)

Task / Stakeholder	Prime Contractor	Sub A	Gov't Test Team
Unit, Integration test creation	X		
Simulation		X	
Validate Acceptance Criteria			X

- Prevents confusion and ensures accountability
- The test development team owns test creation and documentation
- The software sustainment team ensures compatibility with software updates
- Independent test organization owns validation and independent testing

Having boundaries laid out lets everyone know their responsibilities.

Use a Shared CM System with Role-Based Access

All configuration artifacts (code, tests, IaC for environments, and data) should be in a shared CM system such as Git. This ensures that all teams have access to the latest versions while maintaining security and role-specific access.

Best Practices

- Developers have access to test creation repositories (Read/Write)
- Sustainment teams have read access for integration and bug fixes (Note that these sustainment teams should exist and be working throughout the development)
- Independent testers have separate branches for validating or adding their own tests

Implement Versioned Test Suites

Carnegie Mellon University Software Engineering Institute

Tests suites must be versioned alongside software releases to guarantee compatibility between software and their corresponding tests.

Best Practices

- Store test suite changelogs with details on tests that have been created, deleted, or modified
- Create tags or branches for each test suite corresponding to software versions
- Maintain backward compatibility for older software versions as needed

Standardize Test Formats and Metadata

As much as possible, standardized formats should be used for test metadata schemas to ensure tests are understandable, portable, and useable by all parties.

Best Practices

- Include metadata schema such as test ID, purpose, prerequisites, and expected results
- Use a shared naming convention for files and test cases
- Test artifacts should be self-contained, requiring minimal external dependencies
- Test results should be consumable by the reporting tool

Automate Validation and Verification Processes

Carnegie Mellon University Software Engineering Institute

Test validation and execution should be automated, preferably in a CI/CD pipeline to reduce manual effort, increase consistent results, and enabling trust through transparency of test results

Best Practices

- Test teams are responsible for implementing both static and dynamic tests and analyzing their results
- Any independent test organizations should validate tests against defined criteria using CI/CD
- Test artifacts should include execution logs, reports, and validation results all stored in a shared repository. This is often inside the CI/CD system, so appropriate access should be granted.

Maintain Baseline or Common Test Environment

Test environments should be defined and documented to ensure tests are executed consistently across organizations.

Best Practices

- Use Infrastructure as Code (IaC) to standardize environment setup
- Separate configurations should be maintained for development, validation, and production testing
- Environment definitions should be version-controlled along with application code, and tested for reproducibility

Archive and Trace Test Artifacts

Test artifacts should provide traceability to requirements, defects, and releases to facilitate accountability and simplify audits.

Best Practices

- Maintain a traceability matrix linking tests to software features, requirements, or defects
- Environment definitions should be version controlled along with application code, and tested for reproducibility
- Older test suites should be archive and properly annotation on their purpose
- Unique IDs should be used for tests across all organizations

Define Independent Test Validation Processes

Carnegie Mellon University Software Engineering Institute

Independent test organizations should validate existing tests and execute their own tests without compromising integrity of the configuration management. This allows for the neutrality necessary for independent testing and builds confidence in the process.

Best Practices

- All tests must have documented validation criteria
- Results should be shared early and often to all stakeholders, especially developers
- Separate repositories or branches should be used for independent tests

Regular Communication and Feedback Loops

Carnegie Mellon University Software Engineering Institute

Formal communication channels and review cycles should be established and planned across all organizations to promote collaboration and align goals and priorities.

Best Practices

- Create a dashboard to track test results, and give visibility into progress, improvements, and setbacks
- Establish a regular cadence to meet with all stakeholders to discuss testing process and metrics
- Use a ticketing system to track and resolve issues related to test processes and results

Carnegie Mellon University
Software Engineering Institute

Presentation Name

Hardware In The Loop Testing Considerations

HIL Testing

- Modern software development benefits from DevSecOps approach:
  - Development Pipeline
  - Automated testing, including security
  - Continuous Integration/Deployment

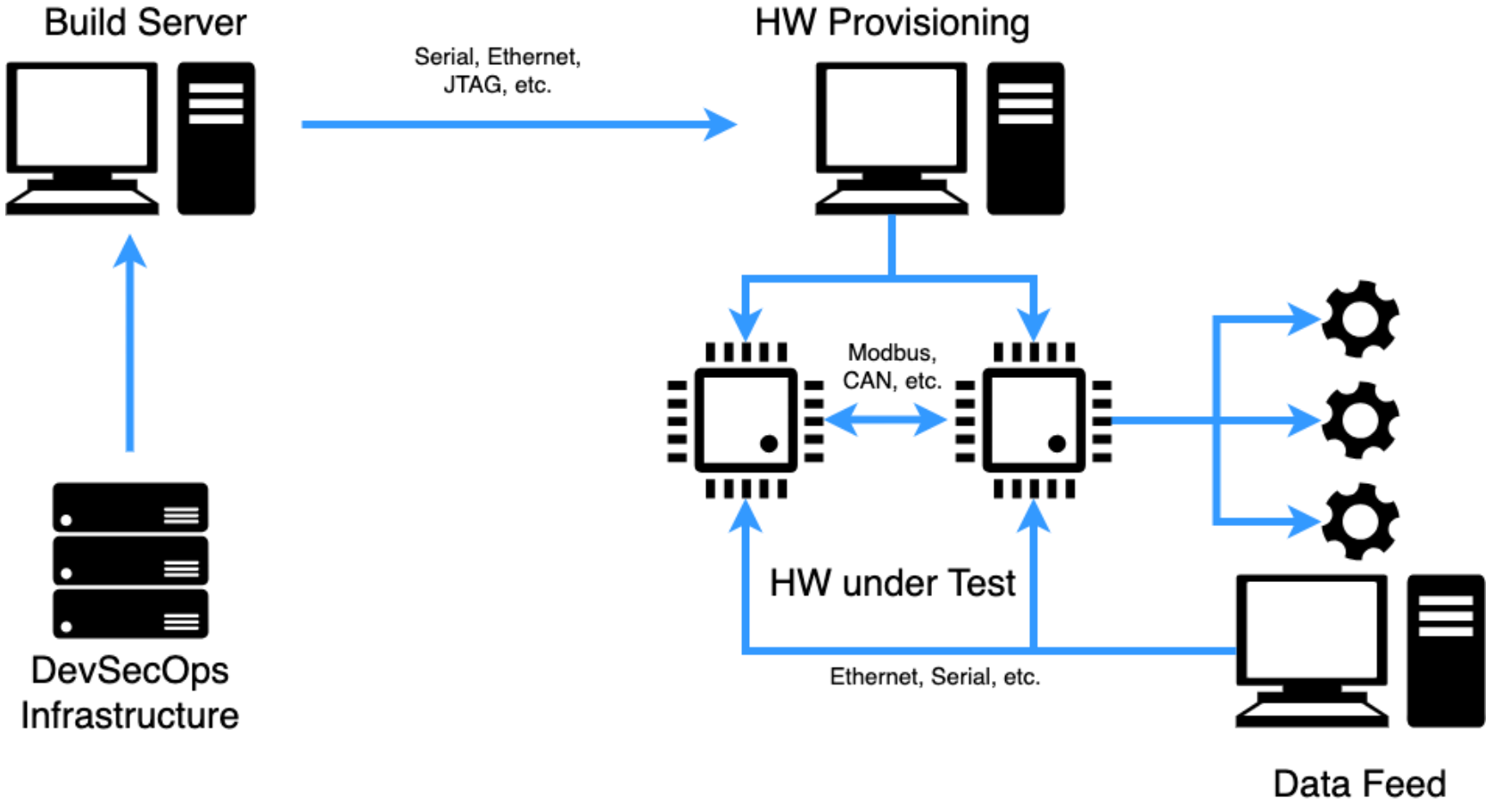
- Much harder to do if hardware is involved:
  - Requires hardware testbeds
  - Complicated and unstable toolchains
  - Unstable I/O to external dependencies
  - Often slow response time

DevSecOps and Hardware Components

- Simulate HW when you can for unit, sub and system tests
  - Don’t expect for simulation to completely replace HW CI
- Don’t wait until end-to-end testing to test with real HW components
  - Perform HW “arming” tests frequently
- HW/SW configuration for reliable CI is very challenging unless simulating
  - Consider full memory snapshots for SW and HW components

Simulated hardware allows tests to be performed quicker.

Hardware-Based Testing



Development boards and prototypes

- Typically used during initial prototyping but can also extend into testing
- Convenient due to multitude of available I/O options already built in
- Very limited as they usually represent the controller, no custom hardware, sensors, etc.
- Not the actual hardware, requires system level tests on actual hardware

Hardware substitution with API

- At the edge of software and hardware
- Hardware replaced in software with a set of API calls
- Closer to simulation but easier to develop
- Not a bad option to do early unit tests on the software side only
- Does not test hardware at all and does not fully test the software either

Hardware Simulation

- Fully or partially implemented hardware functionality in software
- Emulation is often down to the instruction set
- Many forms exist, each has its limitations
- Emulation introduces latency
- Significant effort to create and maintain a functional simulator
  - Many companies have a dedicated simulator “SIM” team
- Efficient development of complex systems “requires” a simulator, often custom
- Off the shelf simulators exist, provide generic simulation and test integration capabilities
  - May not be sufficient for a complex system
  - Proprietary technology maybe difficult to extent

Hybrid Approach

- Start up the simulation team early in the dev process
- Perform early SW development with API-based substitution and HW dev boards
- Basic simulator ready in time for sub-system and unit tests in CI
- Hardware testbed and simulator management with configuration and memory snapshots to improve test stability
- Daily/Weekly “arming” tests with real HW
- End-to-End tests on real hardware once ready

Simulators can be used with CI pipelines and snapshots can make stable tests.

Diving into Simulation

- CPU/Controller emulation at instruction set level
  - Very powerful as it allows snapshots and replays of the entire state of the system
  - By itself, not very useful for complex systems
- Component/peripheral devices emulation
  - Serial interfaces, USB controllers, network, I2C, SPI, Flash, etc.
  - Very important for a reliable simulation
  - Difficult to customize
- Perfect for Kernel or embedded system development
  - Rely primarily on the CPU and I/O virtualization
  - Commercially available CPU virtualization modules
  - Virtualize development boards

Carnegie Mellon University
Software Engineering Institute

Presentation Name

Roles And Responsibilities

Roles and Responsibilities

Carnegie Mellon University Software Engineering Institute

Category	Roles
Program / Product Management	• Portfolio Management • Finance & Budgeting • Program Manager • Project Manager • Scrum Master
Requirements Engineering & Analysis	• Domain SME • Requirements Engineer • System Analyst • Product Owner • Product Manager
Development and Design (Application)	• SW Dev Engineer • SW / Cloud Architect

Category	Roles
Development and Design (Platform)	• SW Dev Engineer (IaC) • DSO Engineer • Site Reliability Engineer • Platform Architect • Platform Engineer
User Design	• User Experience Designer • User Interface
Security	• Security Engineer • CISO • SCA
Test / QA	• Test Manager • Test Engineer • T&E QA Analyst • Test Automation Engineer

Category	Roles
Data Management	• Data Analyst • Data Scientist • Data Engineer • DBA
Operations	• Platform Ops • Application Ops • Network Engineer
Configuration & Release Management	• Configuration Manager • Release Manager
User Support	• Platform Support • Application Support • Tech Writer • Web Site Designer

Testing roles considerations

It is typical for test roles to include the following:

- **Test Manager** - Oversees strategy, planning, and execution; ensures alignment with project goals
- **Test Engineer** - Designs, executes, analyzes test cases to validate software functionality and performance
- **T&E QA Analyst (Test & Evaluation Quality Assurance Analyst)** - Ensures compliance with standards; validates test results and artifacts
- **Test Automation Engineer** - Develops and maintains automated test script, frameworks, and test environments
- **Performance Test Engineer** - Focuses on load, stress, and scalability testing to ensure system reliability
- **Security Test Engineer** - Conducts vulnerability analysis, security assessments, and static and dynamic application security testing
- **Configuration Manager** - Manages test environments, version control, and test data consistency

Make sure to allocate enough resources and identify training necessary to support the various roles of testing.

Test-Driven Development (TDD) requires testing skillsets similar to developer skillsets.

Presentation Name

Software Tools

Carnegie Mellon University
Software Engineering Institute

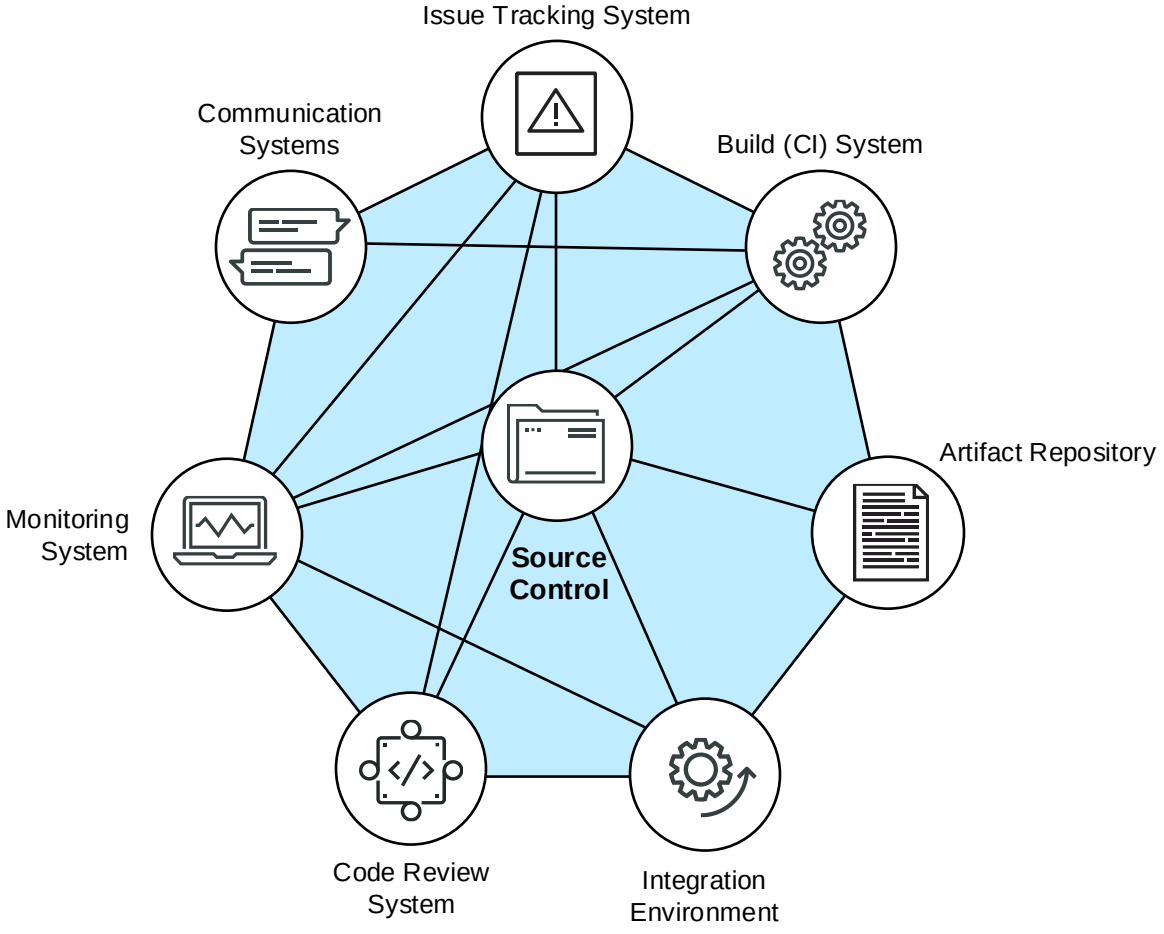
Software Tools

- Test Environment Creation
  - Containerization IaC
    - Docker
  - Virtualization IaC
    - Vagrant, Ansible, Terraform
  - Log viewing
    - new relic, splunk
- Configuration Management
  - Source Control
    - Git, GitLab, Subversion, Bitbucket
  - Issue Tracking
    - Jira, Bugzilla, Xray(Jira add-on)

Test automation

- Frameworks
  - Selenium, Playwright, JUnit, TestNG, Jenkins
- Performance tools
  - LoadRunner, JMeter
- Build automation
  - Maven, Gradle

Integrated Pipeline – With Tooling



- **Source Control**
  - *Bitbucket, Gitlab*
- Issue Tracking System
  - *Jira, Gitlab*
- Build (CI) System
  - *Gitlab, Jenkins*
- Artifact Repository
  - *Nexus, Artifactory*
- Integration Environment
  - *Custom, IaC with Docker, Ansible, Terraform*
- Code Review System
  - *Most integrate via pull request process*
- Monitoring System
  - *Custom tooling/coding usually needed with products like Prometheus, Grafana, etc.*
- Communication Systems
  - *ChatOps, Wiki's, must integrate with other tools*

Contact Us