
# Technical Discussion: Test & Evaluation of Software-Intensive Systems & DevSecOps

Eileen Wrubel – Technical Director of Transforming Software Acquisition Policy & Practice,

Hasan Yasar – Technical Director of Continuous Deployment of Capability,

SEI Software Solutions Division

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

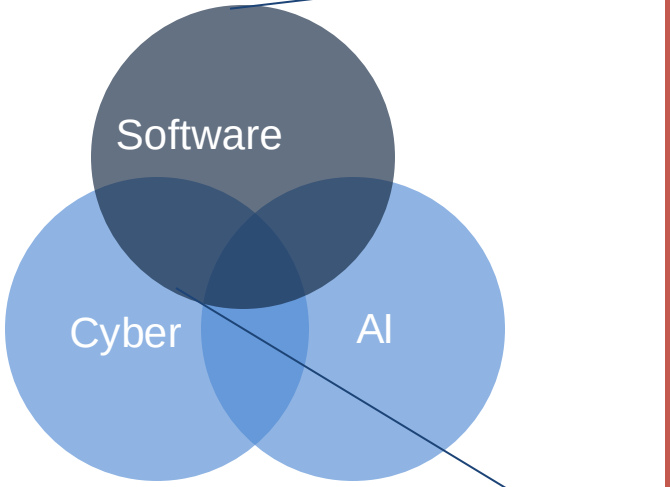


DM22-0934

# Agenda

- **Context for the Software Solution Division**
- **Related project work in DOT&E**
- **Potential collaboration opportunity – an ML component testing challenge**
- **Shift Left Testing & DevSecOps insights**

# Software Solutions Division:
## *Rapidly Deploying Software Innovations with Confidence in DoD*



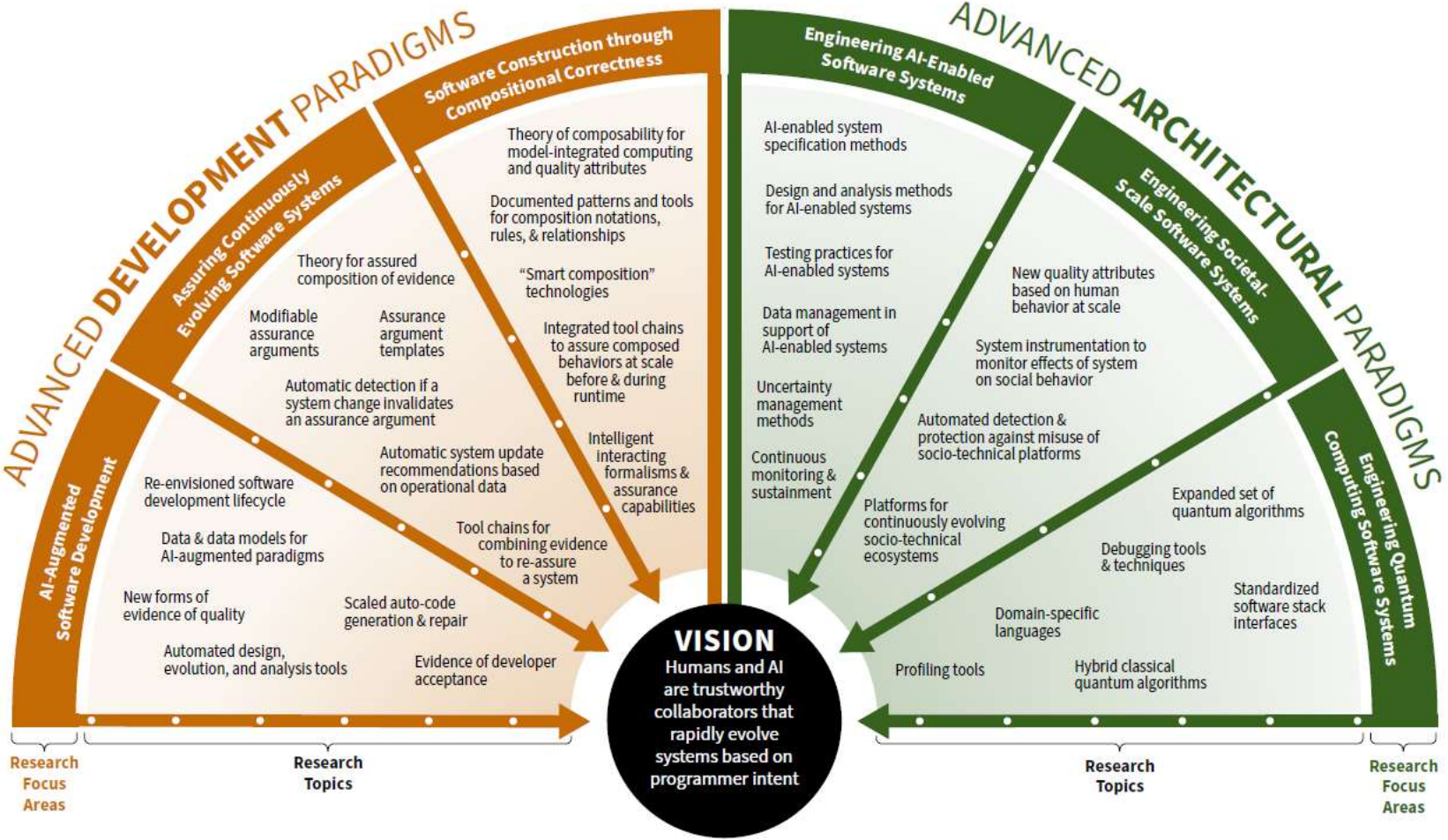

SEI Strategic Framework

Exploit emerging technologies

- Engineering Intelligent Software Systems
- Enabling Mission Capability at Scale
- Assuring Cyber-Physical Systems
- Continuous Deployment of Capability
- Transforming Software Acquisition Policy and Practice

Curate & analyze data for decision-making

# Software Engineering Research Roadmap (10-15 Year Horizon)



# Research Focus Areas: Development Paradigms

## *AI-Augmented Software Development*

The focus of this research area is on what AI-augmented software development will look like at each stage of the development process & during continuous evolution, where it will be particularly useful in taking on routine tasks.

## *Assuring Continuously Evolving Software Systems*

The goal of this research area is to develop a theory & practice of rapid & assured software evolution that enables efficient & bounded reassurance of continuously evolving systems.

## *Software Construction through Compositional Correctness*

This research area focuses on methods & tools that enable the specification & enforcement of composition rules for component-based technologies & platforms that allow both the creation of required behaviors & the assurance of these behaviors.

# Research Focus Areas: Architectural Paradigms

## *Engineering AI-Enabled Software Systems*

This research area focuses on exploring which existing software engineering practices can reliably support the development of AI systems, as well as identifying & augmenting software engineering techniques for systems with AI components.

## *Engineering Societal-Scale Software Systems*

This research area leverages the social sciences to develop new software engineering approaches that enable predictable behavior of software systems consisting of people as system components.

## *Engineering Quantum Computing Software Systems*

This research area will enable quantum computers to be easily programmed, & then enable increasing abstraction as larger, fully fault-tolerant quantum computing systems become available.

# DOT&E Engagement

- April 2020 – December
  - FY20 NDAA Sec 231 – Supported DOT&E in selecting, conducting and documenting Digital Engineering Case Studies
  - Software and Cyber Experts Roundtables
- January 2021 – September 2022
  - Software and Cyber Policy & Guidance: SW Acquisition Pathway T&E Guidance; Cyber T&E Focus Area; Cyber T&E Companion Guide; Software T&E Focus Area; DSO T&E Companion Guide; Cyber Survivability White Paper
- September 2022 – November 2023
  - Continued Work on Software and Cyber Policy & Guidance
  - Integrated Modeling of DSO and Operational Test
  - Reduction of supply chain risk through continuous SBOM monitoring
  - Improving M&S through causal learning (project with NUWC)

# Automation of Mismatch Detection and Testing in ML Systems: Problem Statement

As DoD adopts machine learning (ML) to solve mission critical problems, the inability to detect and avoid mismatches between assumptions and decisions made by different system stakeholders creates delays, rework, and failure in the development, deployment, and evolution of ML-enabled systems.*

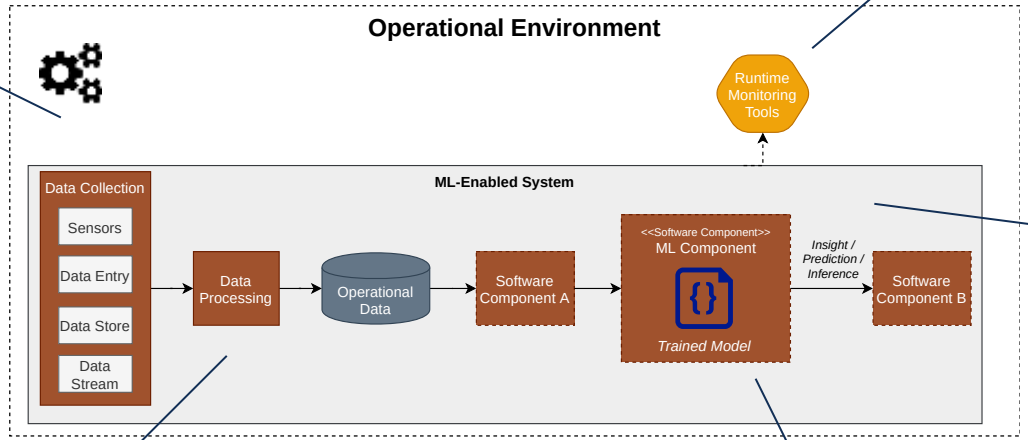
Results from our recent study show that one of the top causes for mismatch is lack of information on how to test ML components.

**ML mismatch** is a problem that occurs in the development, deployment, and operation of an ML-enabled system due to **incorrect assumptions** made about system elements by different stakeholders — data scientists, ML engineers, software engineers, operations — that results in a negative consequence.

* We define an ML-enabled system (or ML system for short) as a software system that includes one or more ML components

# Examples of Mismatch

Poor system performance because computing resources for model testing different from operational computing resources (**computing resource mismatch**)




Tools not set up to detect diminishing model accuracy or collect data necessary for model troubleshooting and retraining (**monitoring mismatch**)

System failure due to poor testing —developers not able to generate appropriate test data or test cases (**testing mismatch**)

Poor model accuracy because model training data different from operational data (**data distribution mismatch**)

Large amounts of glue code because ML component input/output very different from operational data types (**API mismatch**)

# ML Component Testing

Data collected during our research showed that a large cause of mismatch was due to lack of information on how to test ML components, in particular testing for production-readiness, which we define based on four ML component attributes:

- **Ease of Integration (Integratability):** ML component is compatible with upstream and downstream components in production system
- **Testability:** ML component contains metadata/"hooks"/test cases that enable testing by software developers and/or external QA teams
- **Monitorability:** ML component properly produces data (or exposes internals) that is used by monitoring components in the production system
- **Comparable Inference Quality:** Inference quality (e.g., accuracy) of ML component in the production system is comparable to inference quality demonstrated during model development and evaluation

# Near-Term Collaboration Opportunity for ML Component Testing for Production Readiness

Looking for organizations (e.g., T&E) that are tasked with testing ML components (or ML systems) developed by other organizations or teams.

Participate in meetings, calls, or workshops to answer the following questions:

- What are common failures in ML components developed by third parties (or other teams)?
- What type of testing is performed on these components?
- What are the challenges when testing these components?
- What do specifications for ML components look like?
- How do you extract test cases from these specifications?

Longer term potential:

- Testing approaches that we develop for addressing identified challenges
- Integrating SEI-developed approaches into testing processes

# Shift Left Testing with DevSecOps

# Shift-Left testing

Shift left testing requires:

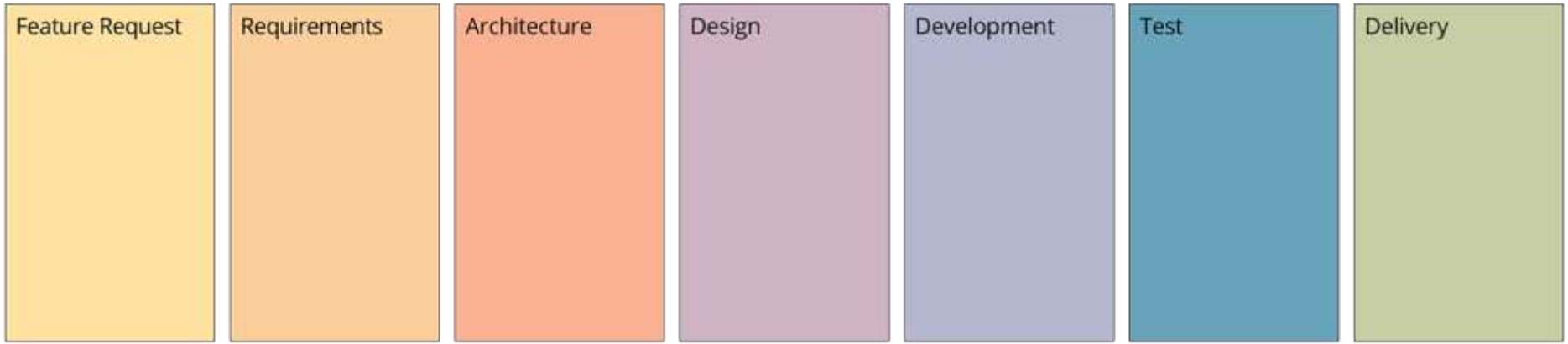
- Practices and test tools
- Timing of tests
- Time to develop automate testing
- Planning for testing(what & how)
- Level of testing
- **Culture**

None of these changes will be easy;

they require *energy, commitment, resources and mindset*

# Testing Fundamentals

# **Reminder**: SW Development Phases



# Different Incremental Approaches

Incremental development:

- Single increment of work, delivered once in a single package

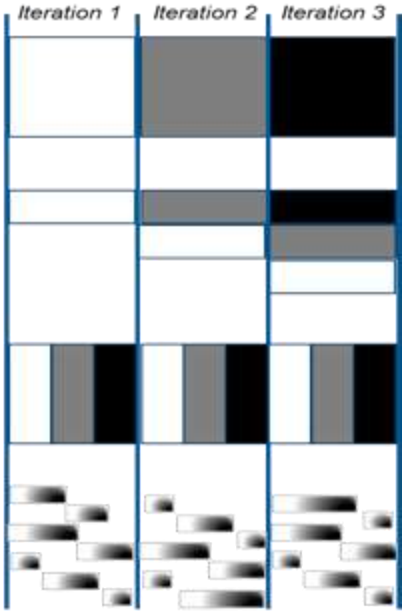
Incremental Development, Single Delivery

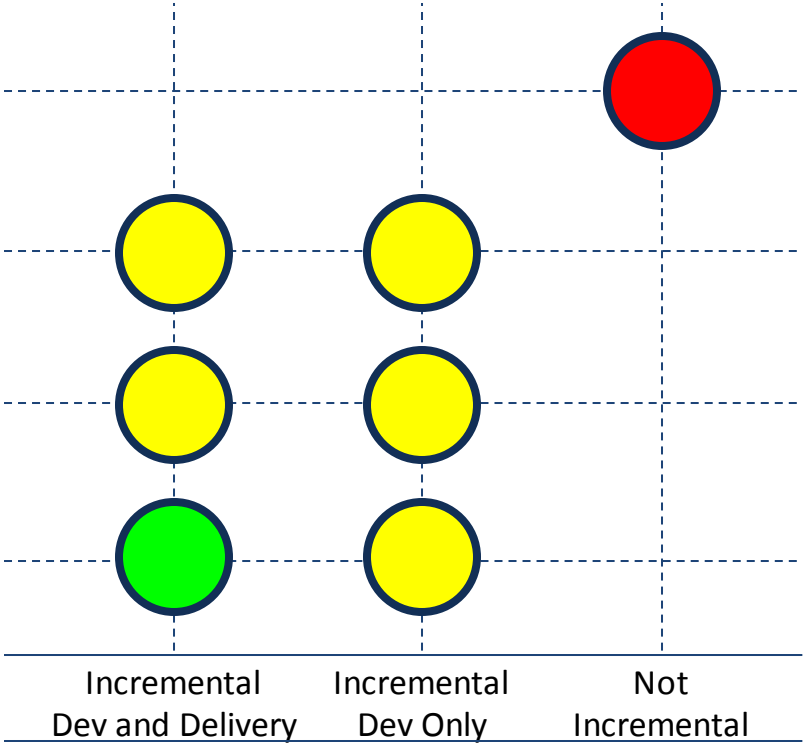
- Work divided into logical subsets for development in pieces, delivered once in a single package

Incremental Development & Delivery

- Work divided into meaningful slices of the total end result, delivered in gradually more complete versions
- Alternatively, delivering new pieces rather than total new versions

# Incremental and Iterative Combinations



# Start from Planning (capturing the right requirements)

## Traditional

- The requirements form a mutually exclusive and collectively exhaustive expression of the user needs and wants
- Complete. Each requirement must fully describe the capability to be delivered
- Unambiguous. All readers of a requirement should arrive at a single, consistent interpretation of it
- Verifiable. It should be possible to objectively determine whether the system properly implements each requirement
- Consistent. A requirement must not conflict with other requirement

## Good User stories (INVEST)

**I**ndependent. The requirement can be developed and tested on its own

**N**egotiable (Definable). The requirement is a promise to have a conversation in due time to define the details of whatever is being built. Is more about learning than negotiation

**V**aluable. The requirement must provide a benefit the customer could appreciate

**E**stimable. It should be possible for the team to forecast the effort it will require to implement it

**S**mall. The requirement should be small enough to be able to be completed in an iteration

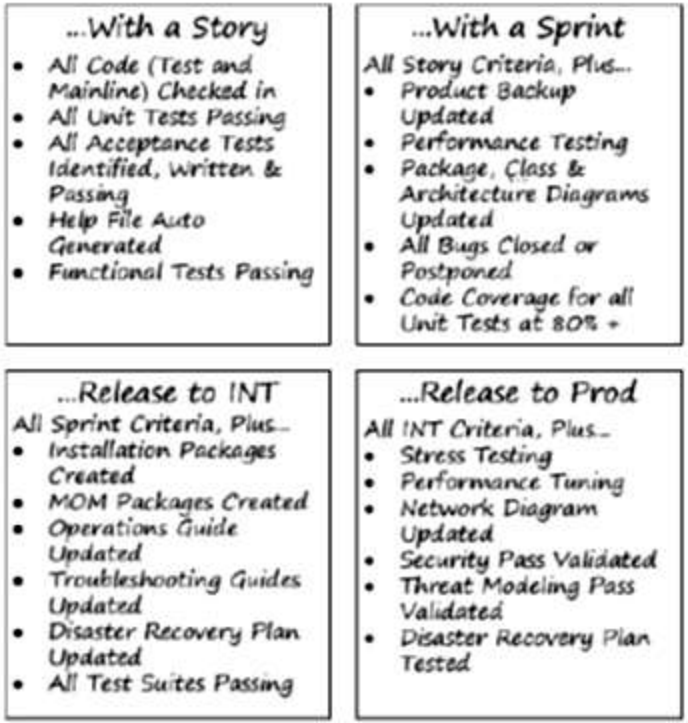
**T**estable. The requirement must provide enough information to make it clear how it will be verified

# Definitions of Ready & Definitions of Done

## DoR

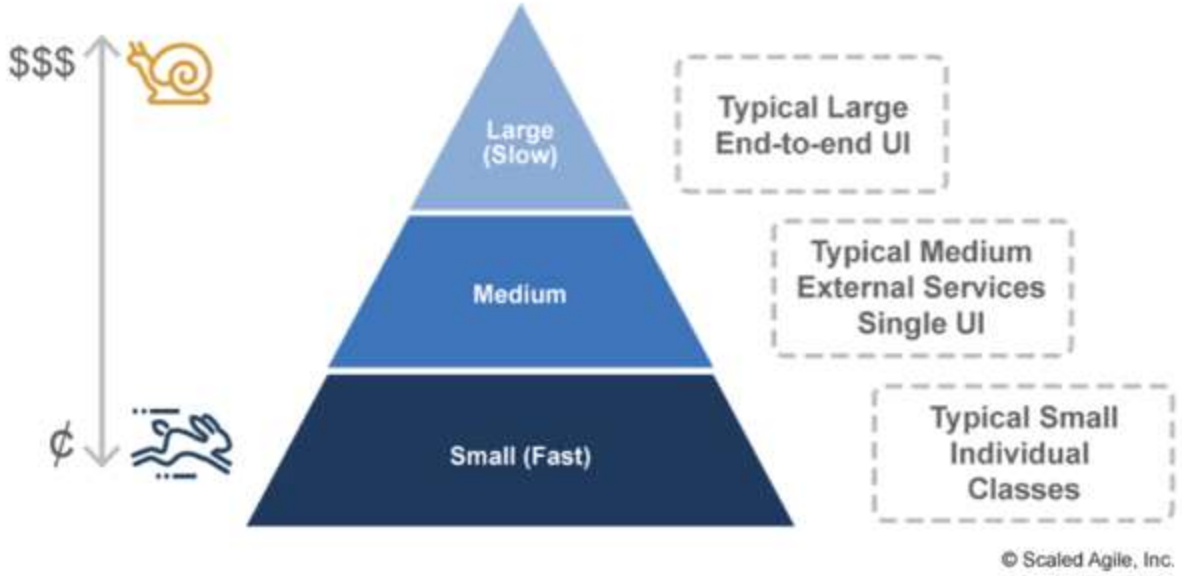
- Business value is clearly articulated
- PBI details are understood by the development team
- Dependencies are identified and resolved (e.g., not known external dependency should block the work once started)
- The PBI is estimated and small enough to comfortably fit in one sprint
- Acceptance criteria are clear and testable.
- Performance criteria, if any, are defined and testable

## DoD




M. Lacey, How Do We Know When We Are Done?, 2008

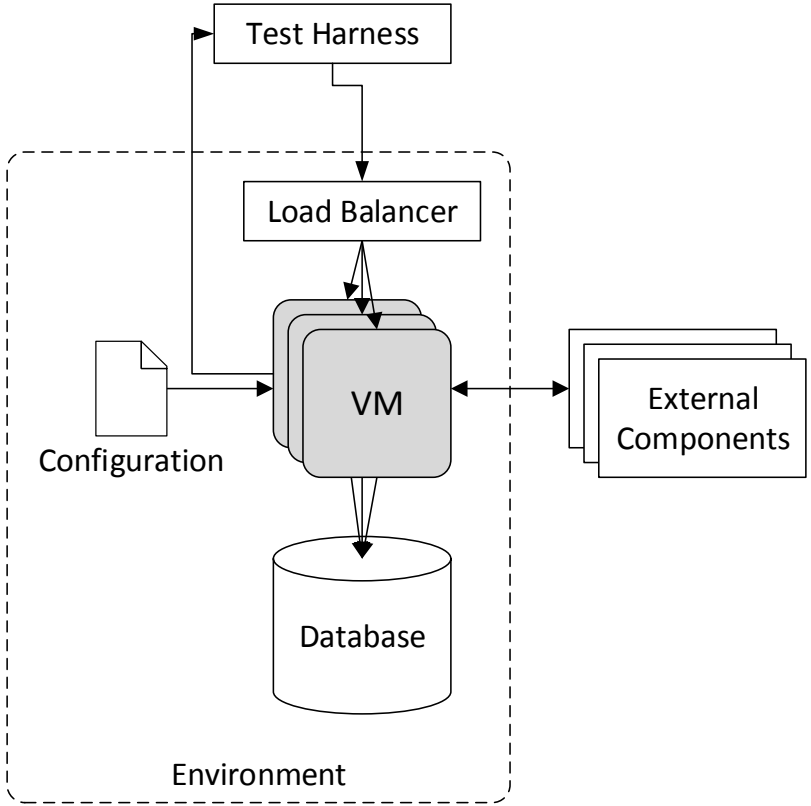
# Testing in General




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

# Test Harness

A test harness generates inputs and compares outputs to verify the correctness of the code.



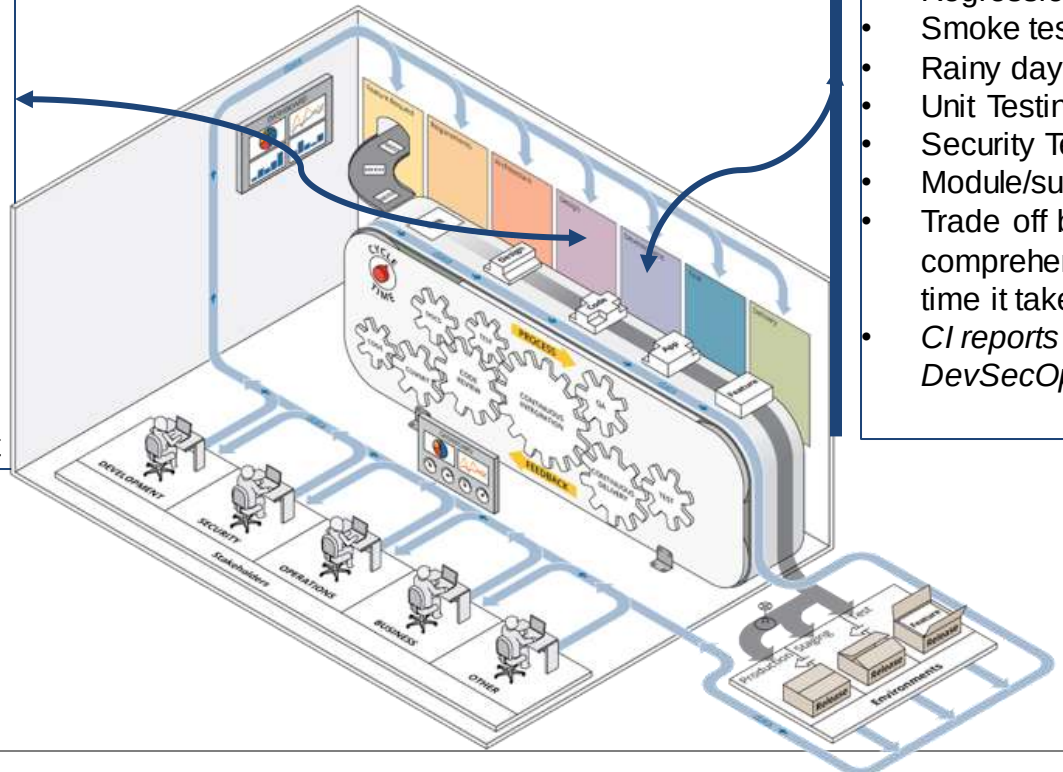
# Test data

Test data is real (ish) but limited.

- Need to worry about exposing private data to Developers/Contractors
- Limited so that tests will run fast and keep build queue from growing
- If tests change database, it must be refreshed before next set of tests.
- No results should be sent to any production service – results in corrupting production version.

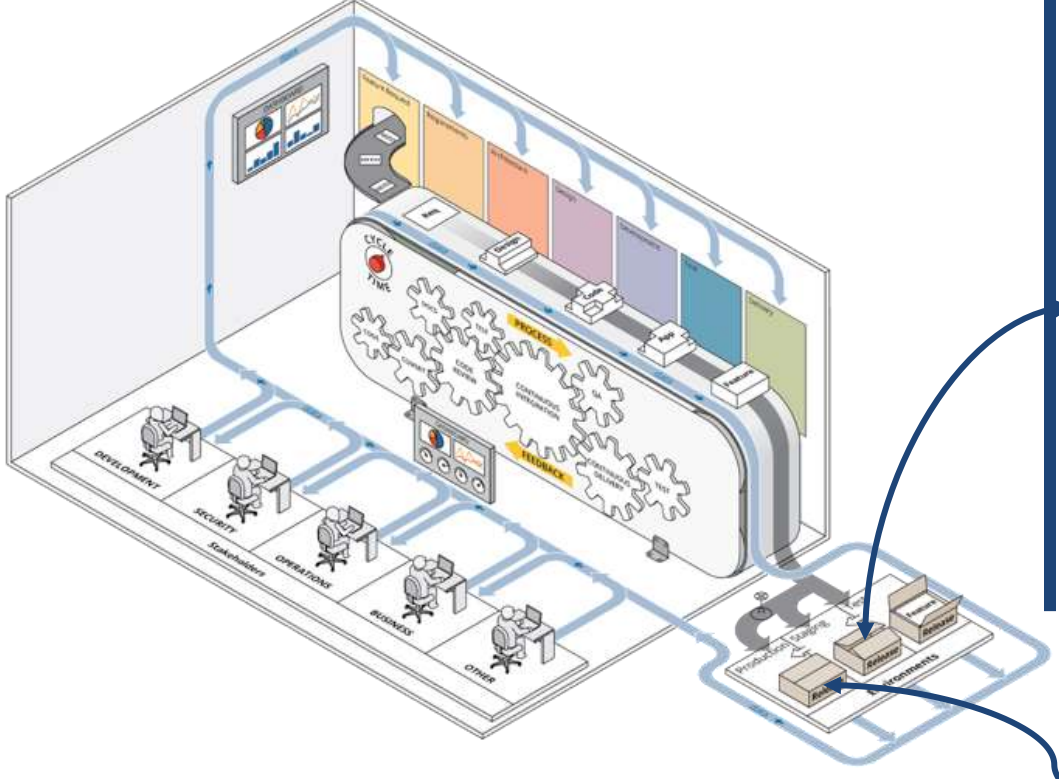
# Testing in Development

- Unit Testing/component testing
- Verify the functionally of specific section of code
- Includes static code analyzers, data flow, metrics analysis or peer code reviews
- Acceptance Testing
- Done by developer prior to integration or regression
- Security Review/Testing
- Reasonable to proceed with further testing or not




# Automated Testing with CI

- Product artifacts are subjected to a collection of automated tests
- Tests are performed using a test harness
- Regression tests
- Smoke testing
- Rainy day scenarios
- Unit Testing/component testing
- Security Testing
- Module/subsystem integration
- Trade off between comprehensiveness of tests and time it takes to run the tests
- *CI reports results through DevSecOps pipeline*



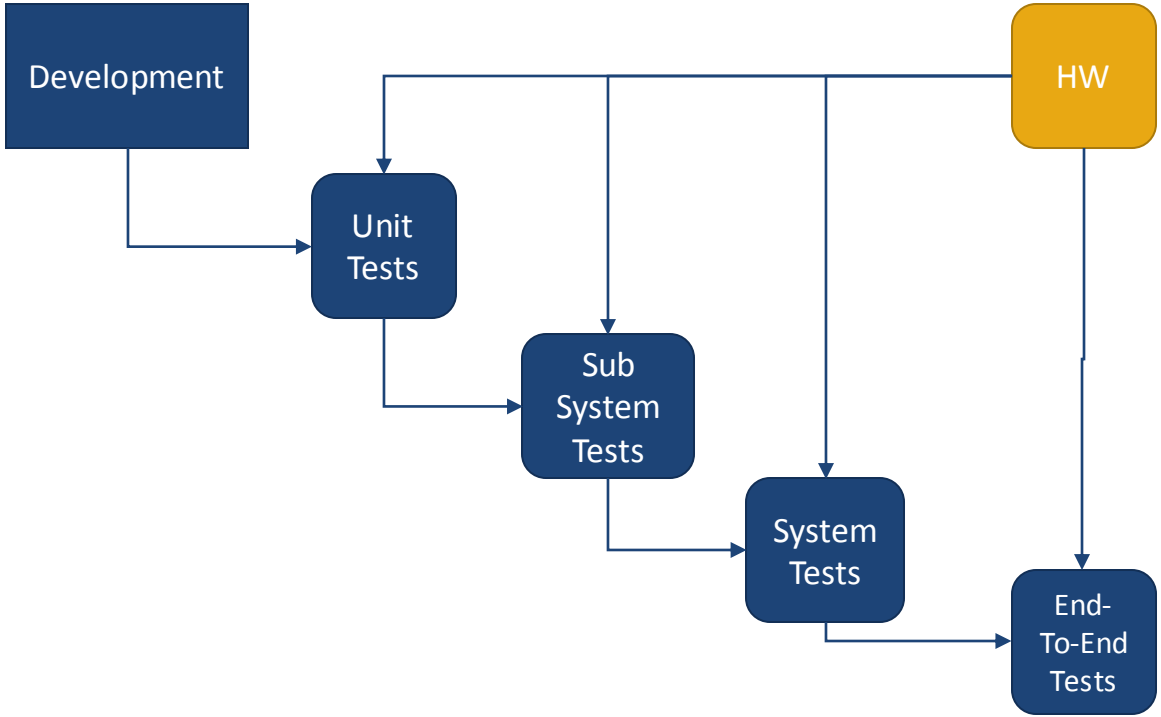

## Testing in Staging.

- Regression testing
- Smoke testing
- Compatibility testing
- Integration testing
- Functional Testing
- Usability Testing
- Install/uninstall testing
- Performance testing
- Security testing
- Test for performance with generated loads
- Environment should be as 'real' as possible
- Load balanced
  - Auto scaled
  - Multiple instances

## Testing in Production(!)

- Performance Testing
- Usability
- Chaos Monkey testing
- Security testing
- Feature flag

# Testing for Embedded systems & HW



# Development boards and prototypes

- Typically used during initial prototyping but can also extend into testing
- Convenient due to multitude of available IO 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

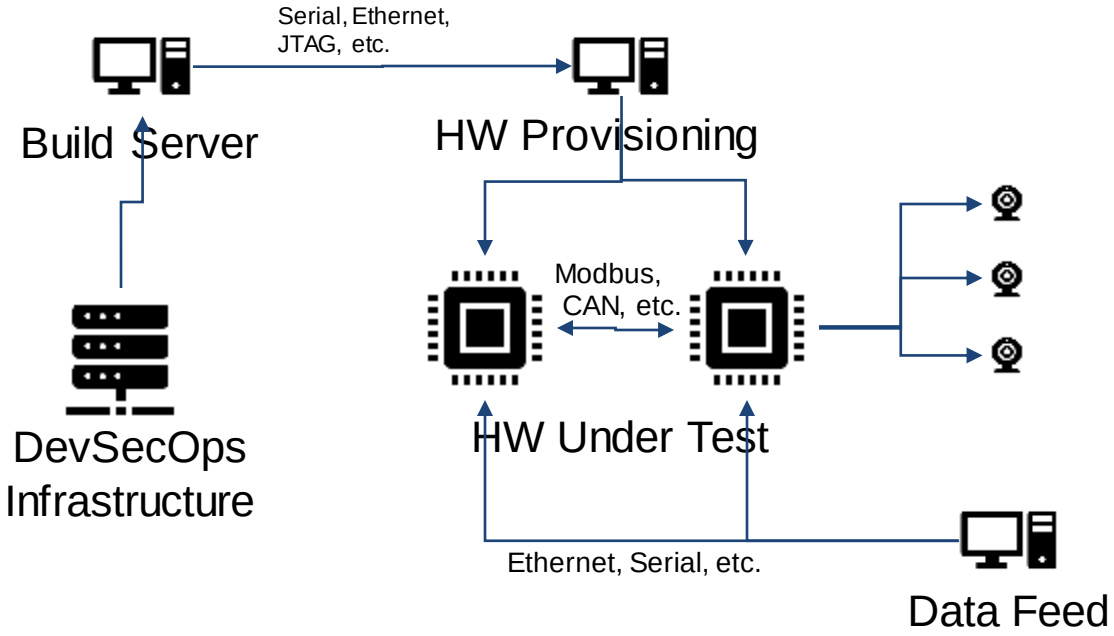
# Hardware simulation

- 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

# Hardware-Based (HWIL) Testing with DevSecOps

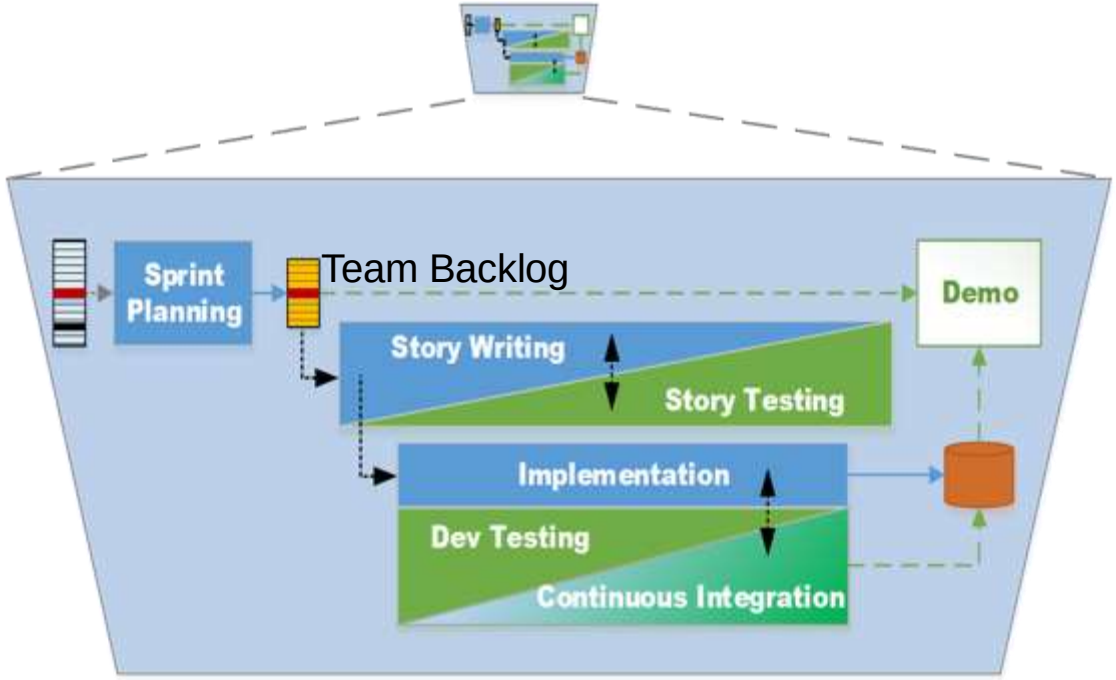


# Overall: View of Testing

- Tests are the final requirements for the system!
  - We use tests to determine if the software is what we want and have become synonyms for the requirements
  - They're the only objective measure of the system
- Consequences:
  - Every requirement **must** have one or more tests associated with it
  - If we're practicing TDD, then code should only be written to make a failing test pass
  - Tests will have to be developed incrementally (or we're back to waterfall)

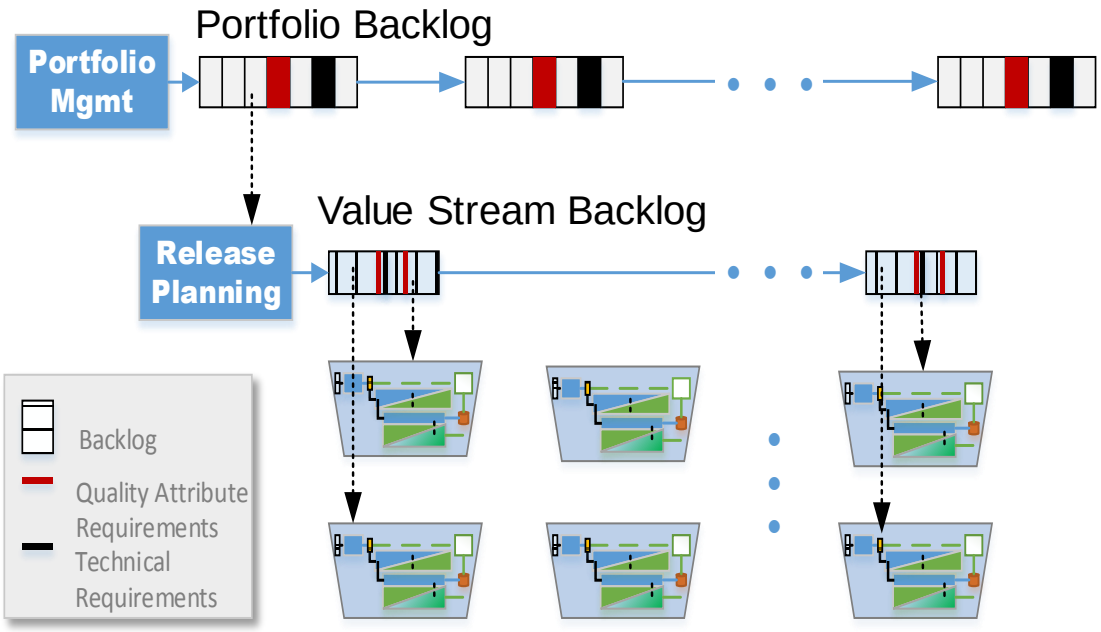
# Agile intermingles Development and Test

- Development and test are not separate or standalone
- Tests are added to a repeatable test suite
- Test suite is repeated for each change
- Demos ***not*** a replacement for testing

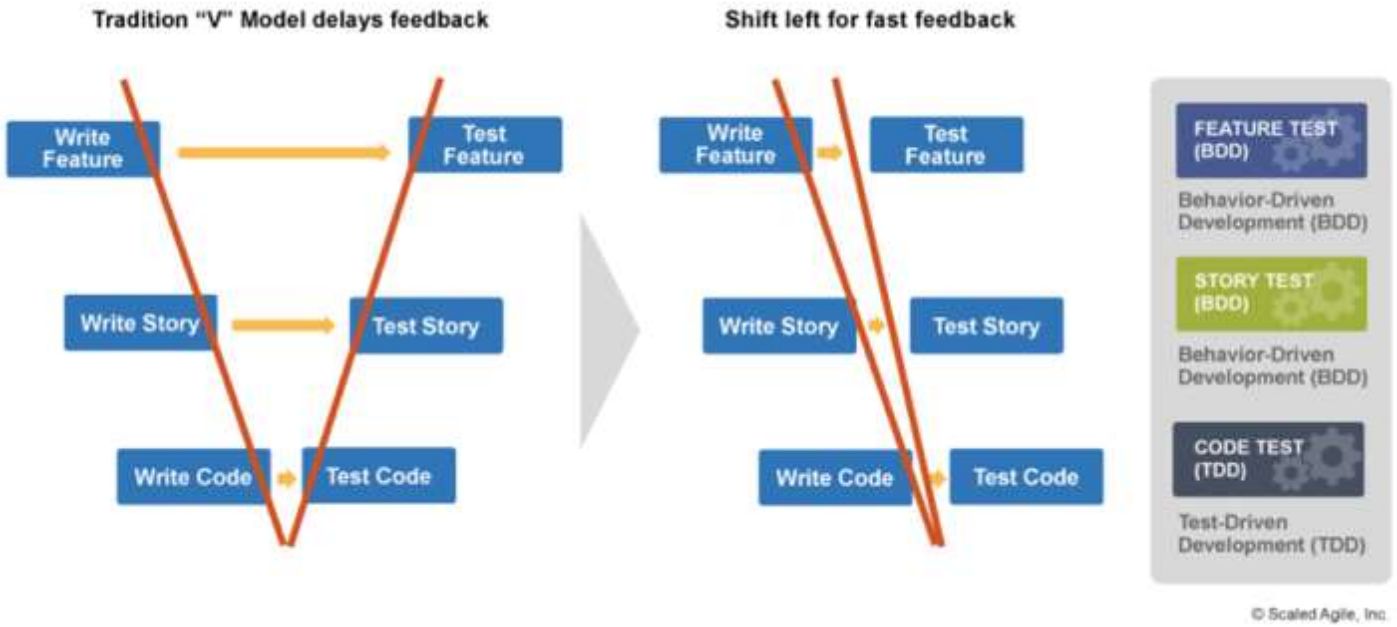


# Agile iteration relies on more testing earlier

- Testing used for rapid refinement of loosely understood requirements, architecture, and design
- Requirements and implementation are finely sliced
- Each slice is tested immediately and repetitively



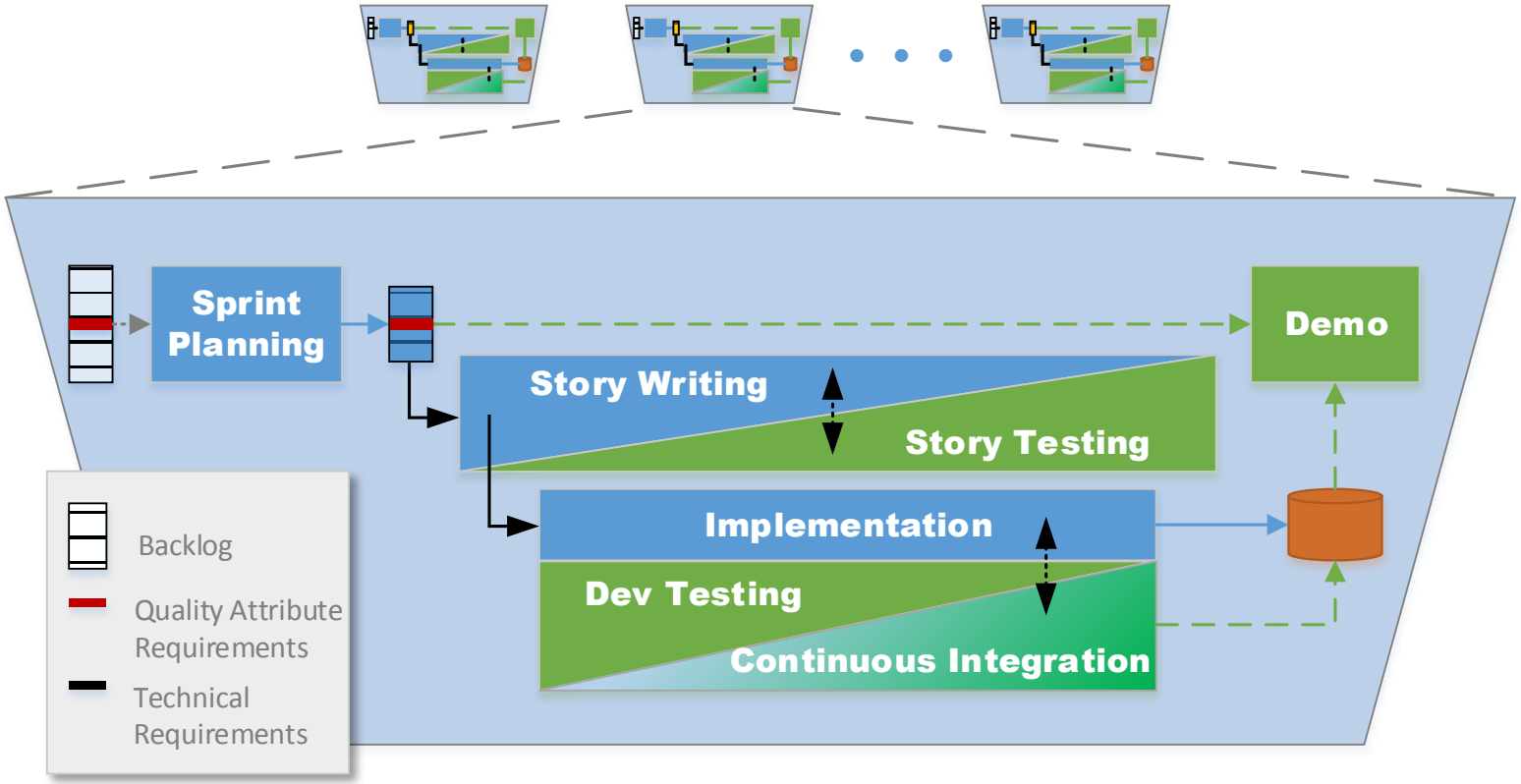
# Shift-left testing



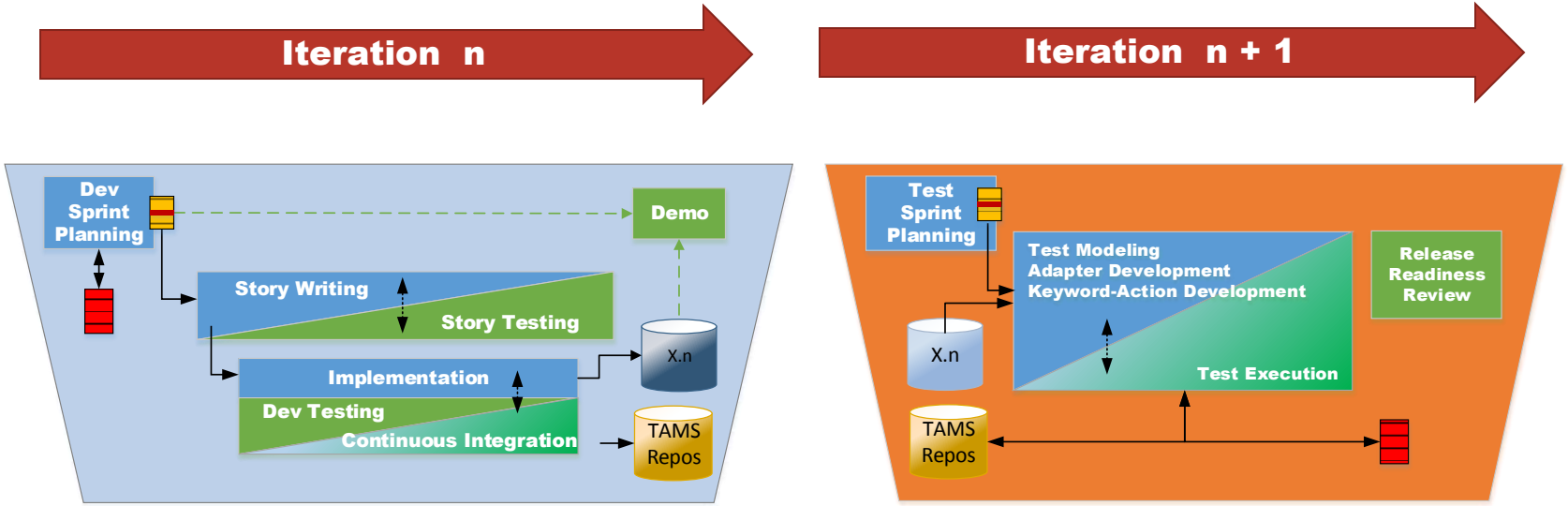

We may not be able to run the tests, but we're thinking about them early

# A Strategy for Shift-Left Testing

# Classic Agile inter-twingles development and test



# Agile High Assurance Dev/Test cadence




Development Sprint – Development Team
Version *x.n* developed and stabilized

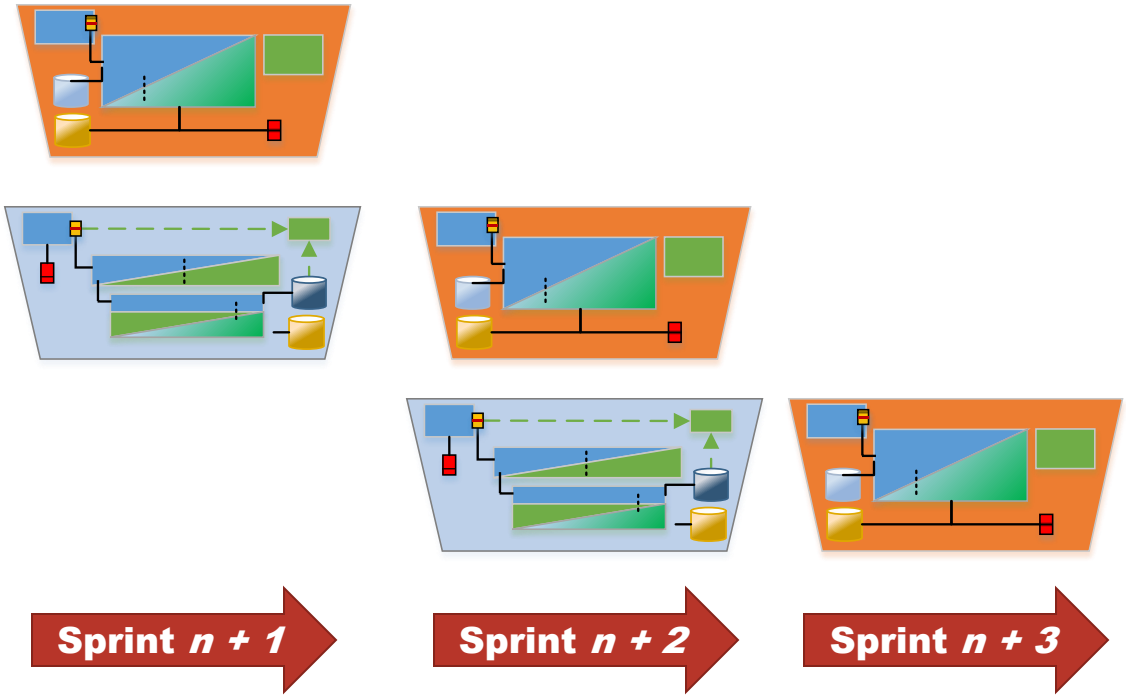
Test Sprint – Test Team
Version *x.n* modeled and stressed

# Agile High Assurance Dev/Test cadence

**Version 0.1**

**Version 0.2**

**Version 0.3**



# Agile High Assurance Dev/Test Cadence

2-4 week Iteration/Sprint

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | Release |
|---|---|---|---|---|---|---|---|---|
| 0 | Release Planning | Test Env Prep | | | | | | |
| N | | Dev 0.1 | Test 0.1 | | | | | |
| N+1 | | | Dev 0.2 | Test 0.2 | | | | |
| N+2 | | | | Dev 0.3 | Test 0.3 | | | |
| N+3 | | | | | Dev 0.4 | Test 0.4 | | |
| N+4 | | | | | | Dev 0.5 | Test 0.5 | |
| Release Candidate | | | | | | | Dev RC | Test RC, Release |
| | | | | | | | | Support, Retro |

Version

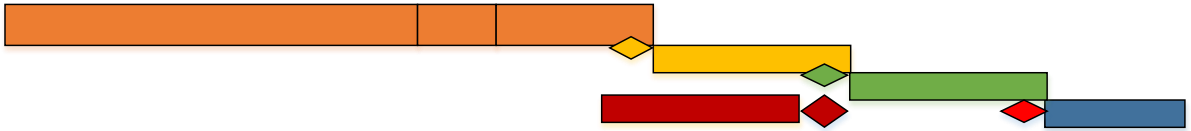
Team Sprint

Dev Sprint

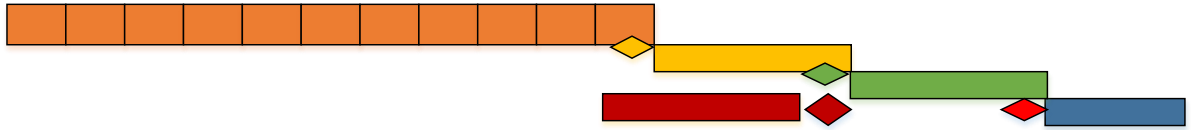
Test Sprint

# Shift left Testing in multiple sprints

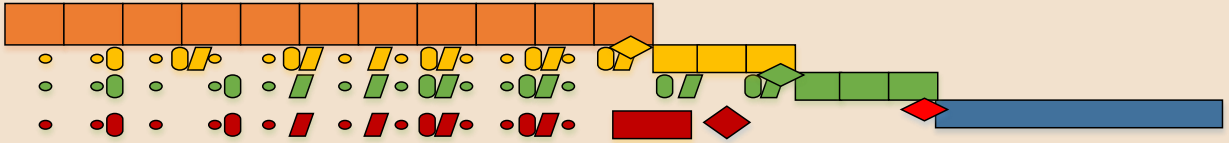
Traditional Vee-process



Agile development with traditional DT and OT (Hybrid)



Agile development with traditional DT and OT, early integration synch points



**Agile High Assurance Dev/Test Cadence**

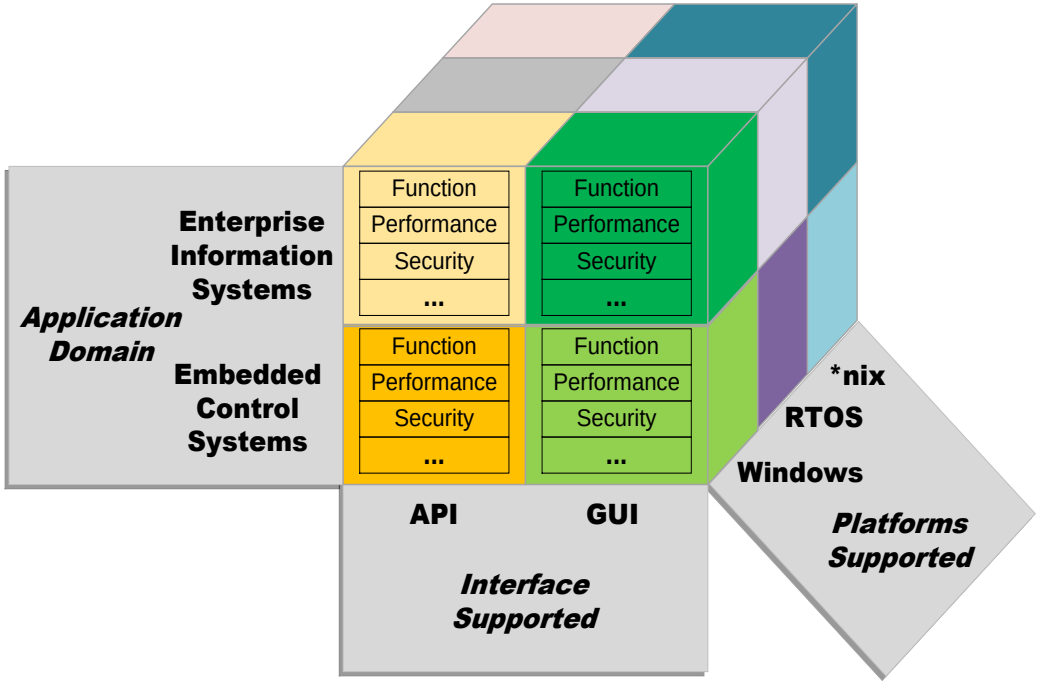
**Moving from phased and siloed testing to Agile testing is the "Big Deal"**

**Integrating Agile cadence with DT/OT is a key challenge**

OEM Dev | Developmental Testing (DT) | Cyber | Operational Testing (OT) | Deployment

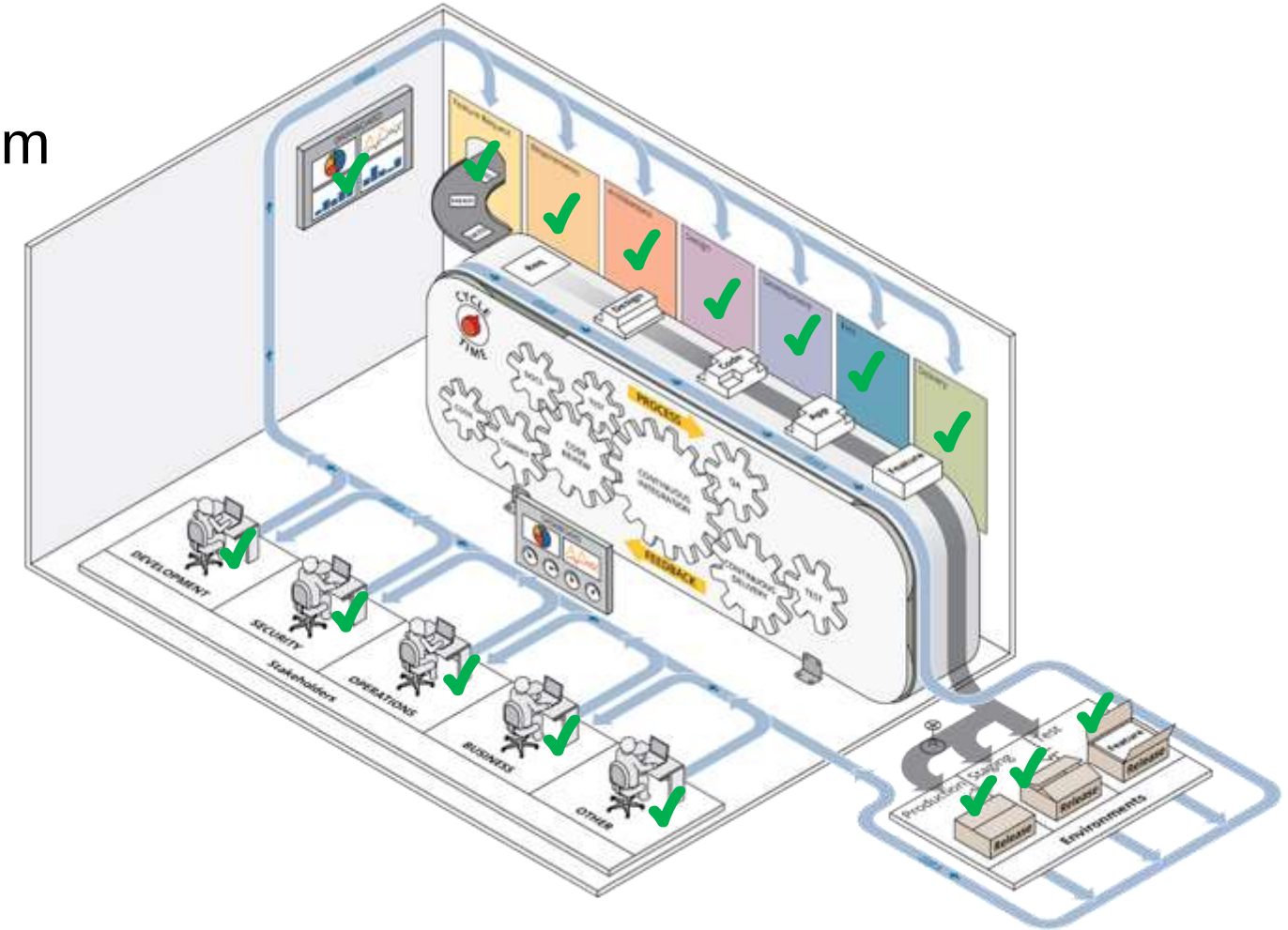
# Tools for Test Automation

- There are *hundreds* of COTS, FOSS, and GOTS software testing tools
- Each tool is specialized for a certain kind of testing
- Each tool is specialized for a tool stack, target stack, and target interface

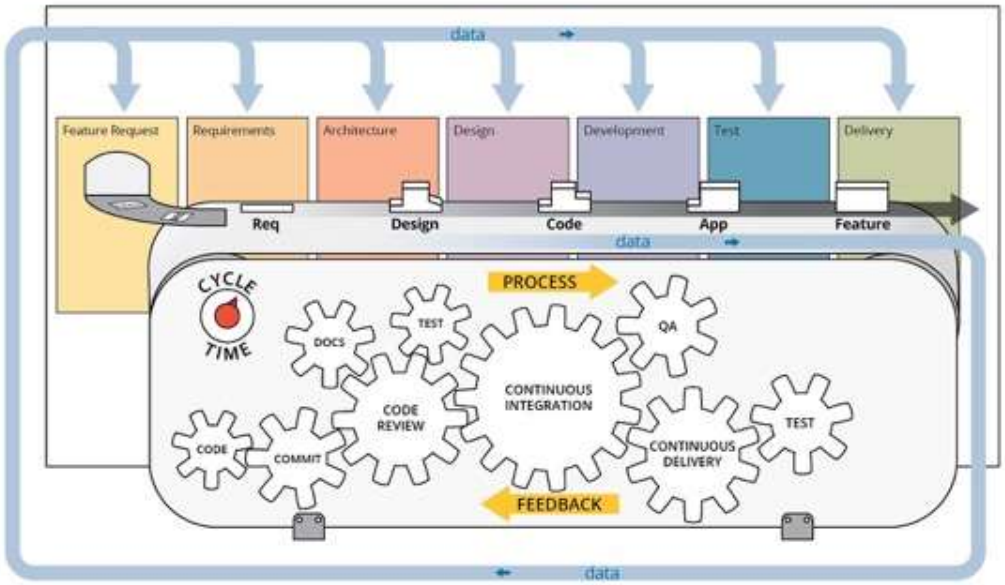


# Takeaway

Apply Testing from *Inception* to *Deploy* and improve every delivery



# Next Steps




- Bring tester community into development early
  - Ideal time is in backlog grooming; testers can develop acceptance criteria
  - Particularly true for SAFe Features and Capabilities – use BDD to define acceptance
- No surprises
  - No hidden tests – provide tests to developers as soon as you have them
  - Independence (e.g., of OT&E) is not isolation
- Integrate. **Integrate. INTEGRATE**
- Automate as much as possible
- Tests should be delivered with the code

# Thank you

**Eileen Wrubel**
*eow@sei.cmu.edu*

**Hasan Yasar**
*hyasar@cmu.edu*