Research Review 2018

Modern Software Lifecycle Practices

Dr. Ipek Ozkaya, Principal Researcher

Dr. Sam Procter, Researcher

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

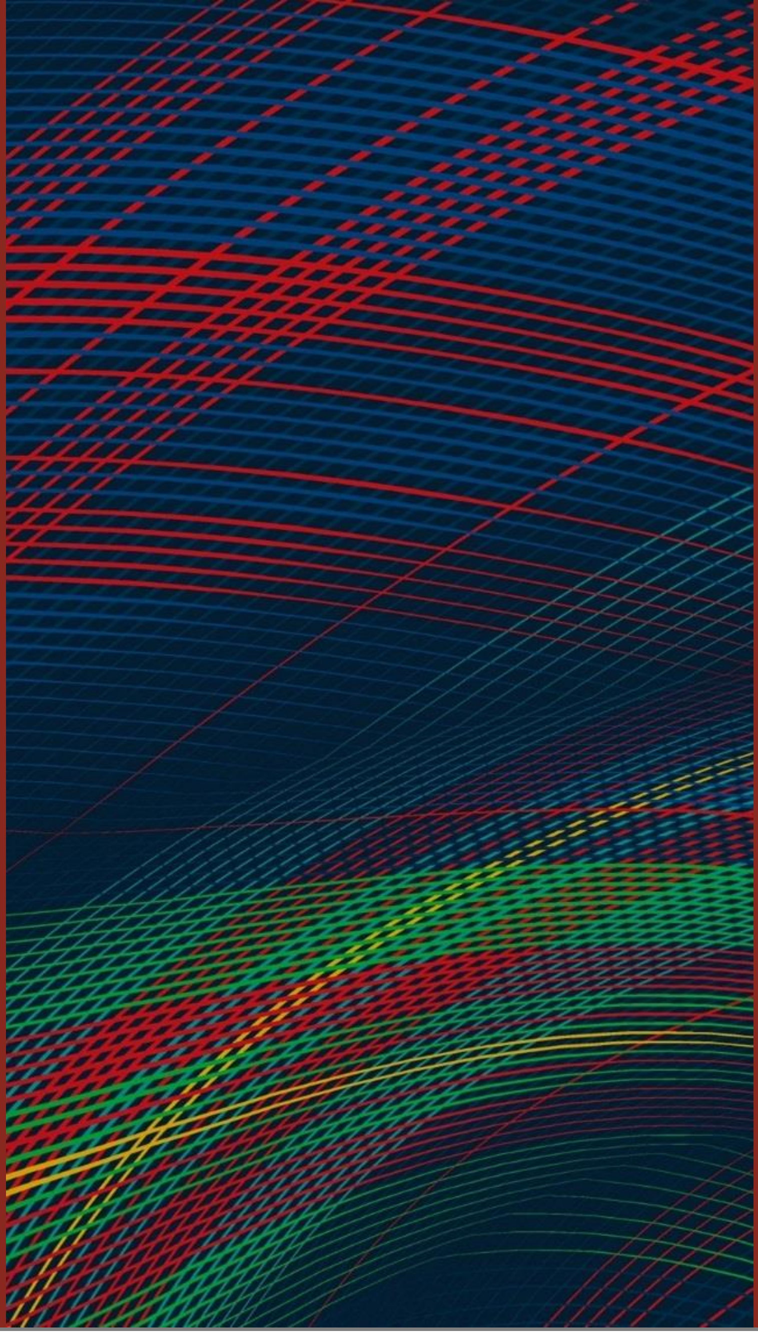
This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM18-1197

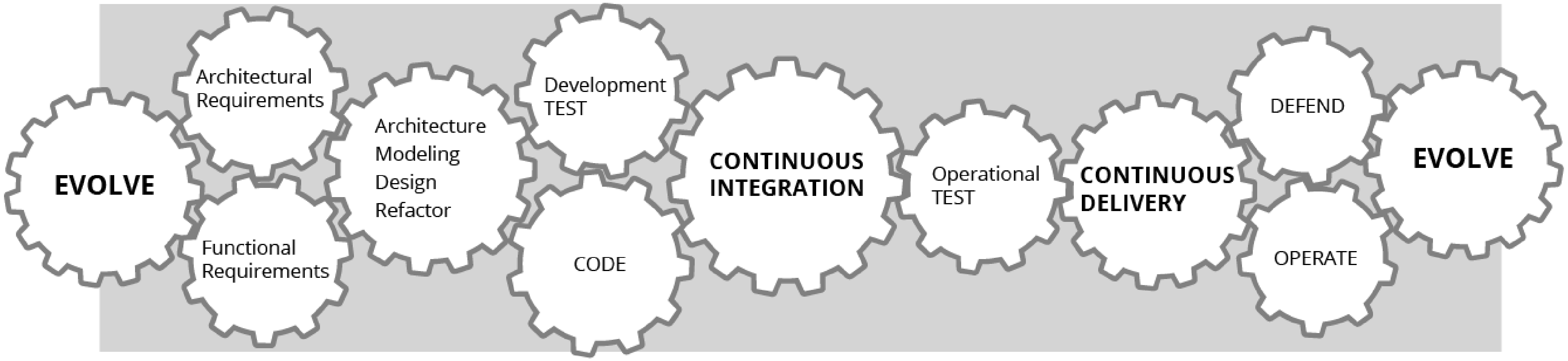
DoD Priorities

Executing system development on schedule and cost

Rapidly evolving systems to exploit new technologies



DoD Challenges in Modern Software Lifecycle Practices



- Manual practices do not scale
- Selecting best-fit software development and analysis methods is not trivial
- Software process metrics do not reflect product realities
- Critical qualities like safety, security, and sustainability are afterthoughts
- Evolving legacy software with new technology is time-consuming, costly, and error-prone

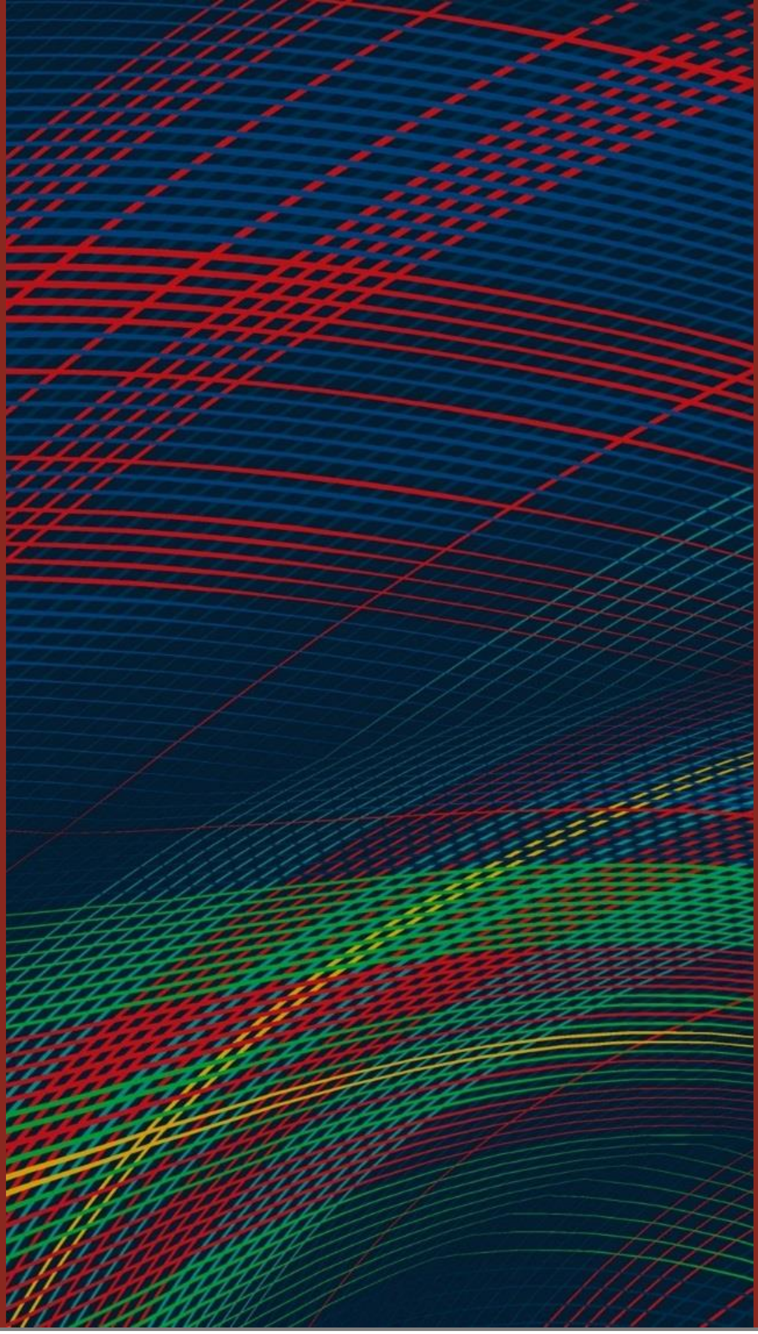
The SEI Approach

Increase automation to enable repeatable, scalable software lifecycle practices:

- Apply model-based techniques to enable safe and secure system development while decreasing uncertainty
- Implement tool-supported system analysis to collect reliable data and enable just-in-time response to problems
- Develop software data analytics, such as applying AI techniques to software data, to improve decision making
- Rapidly pilot interim results with government, industry, tool vendors, and industry partners

Research Review 2018

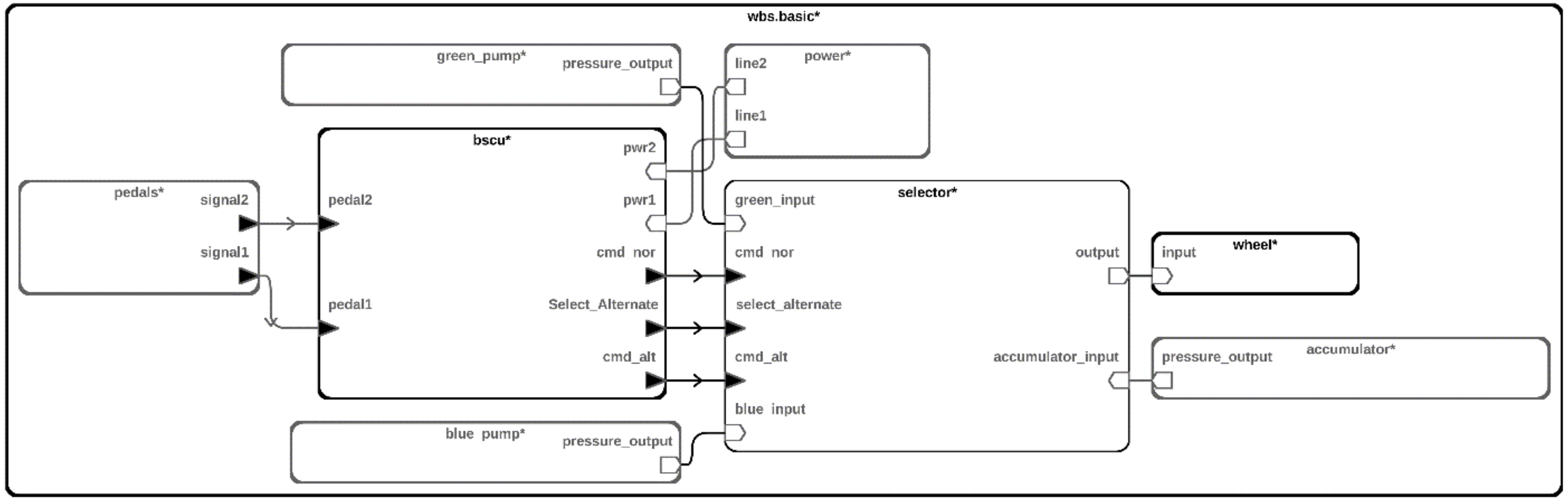
Integrating Safety and Security Engineering for Mission-Critical Systems



DoD Challenges in Critical-System Safety and Security

Problem: Modern safety-critical systems are created by networking heterogeneous components together

State-of-the-art functionality now often requires exposure of those networks to the outside world, so security has become a concern



How should security analysis and design techniques be integrated with their safety-focused counterparts?

Solution: AADL is an internationally standardized architecture modeling language with a 15-year history of successful use in commercial, industrial, academic, and military applications

AADL excels at analyzing component-based systems by

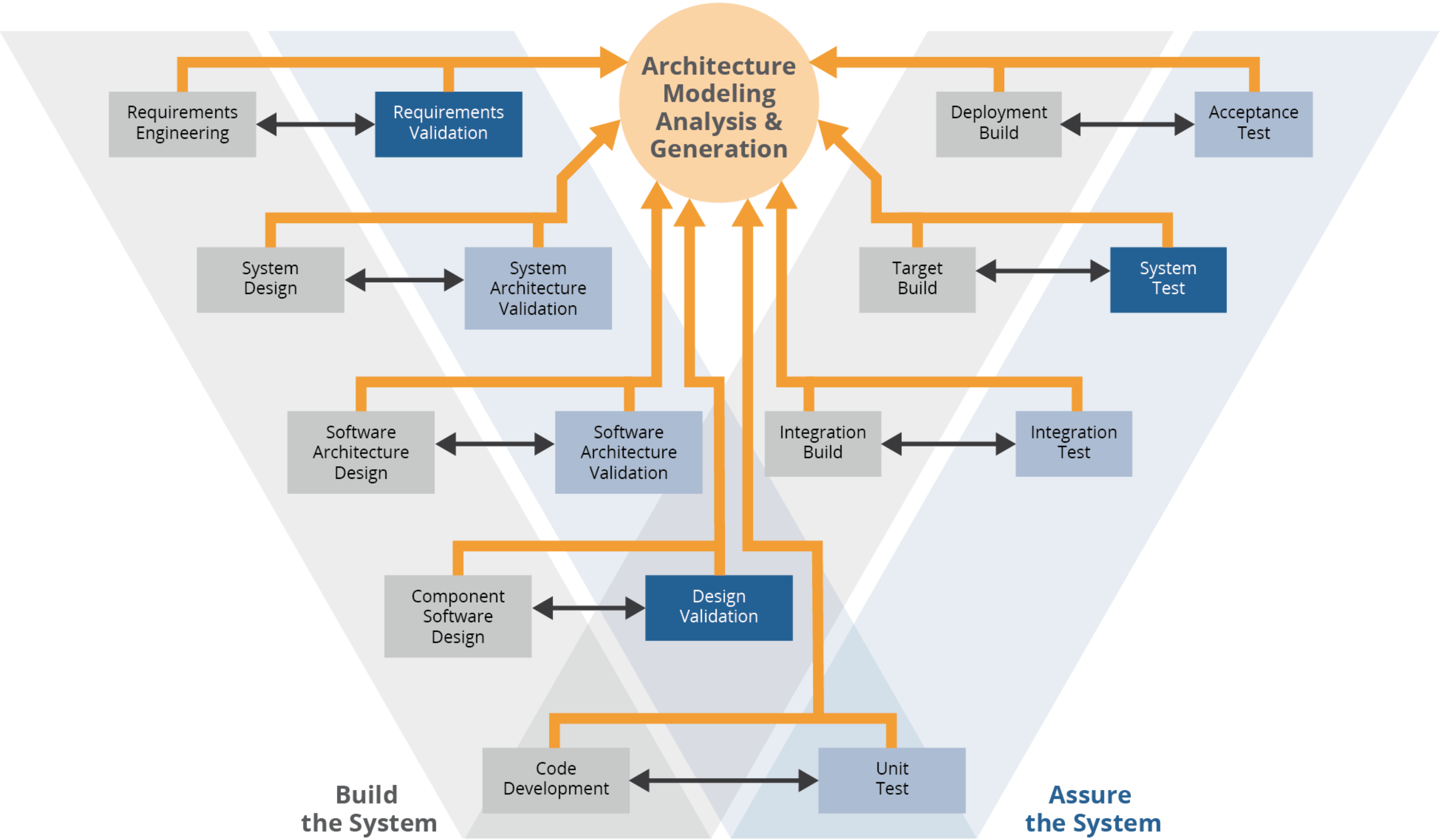
- integrating annotated components
- running system-level analyses

Safety and Architecture

Problem: Safety problems are created early and caught late

Solution: ALISA is a tool kit and process for addressing system safety at the architecture level

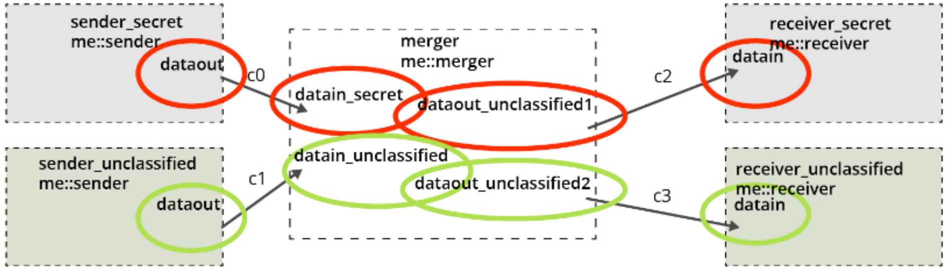
Safety is evaluated in the same way as other quality attributes: components are annotated, and then the integrated system is analyzed



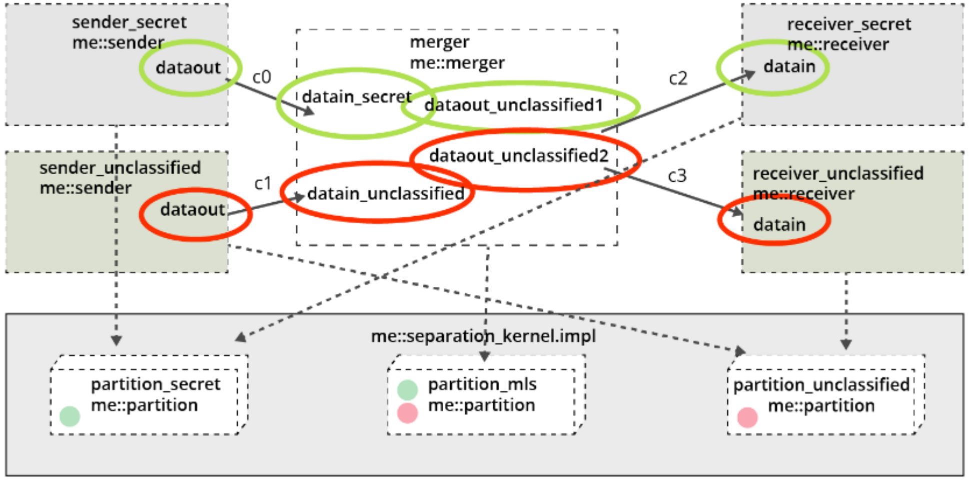
Source: “Architecture-Led Safety Process,” CMU/SEI-2016-TR-012

Security and Architecture

Security policy vulnerabilities:
Analyze information flows



Security enforcement vulnerabilities:
Analyze deployment mechanisms



Previous security architecture research, such as the Multiple Independent Levels of Security (MILS), focused on separating security policy and enforcement

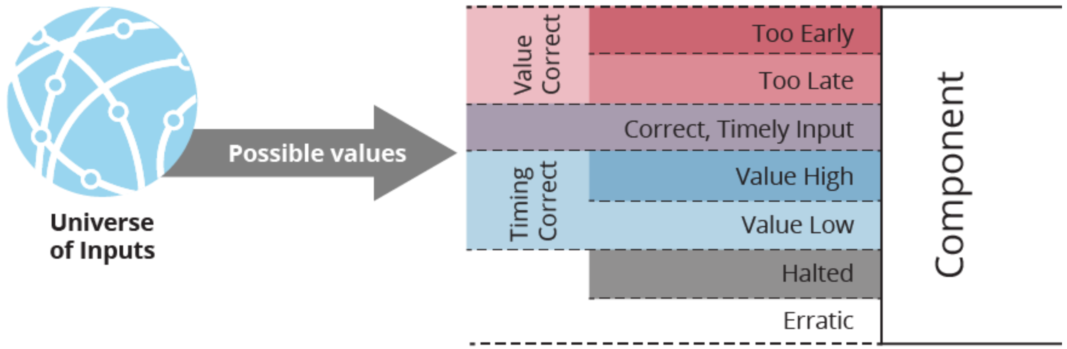
We added support for MILS within AADL/OSATE as well as code generation from models with security policies:

1. Security policy specification
2. Security policy enforcement
3. Generation and deployment of compliant systems

Modeling Security Requirements in the Context of Safety

Approach: Use effects-focused analysis and tooling

- When are various techniques appropriate?
  - Biba model (integrity)
  - Bell–LaPadula (confidentiality)
- What “building blocks” should be used?
  - examples: encryption, partitioning, checksums
- How should requirements be verified?



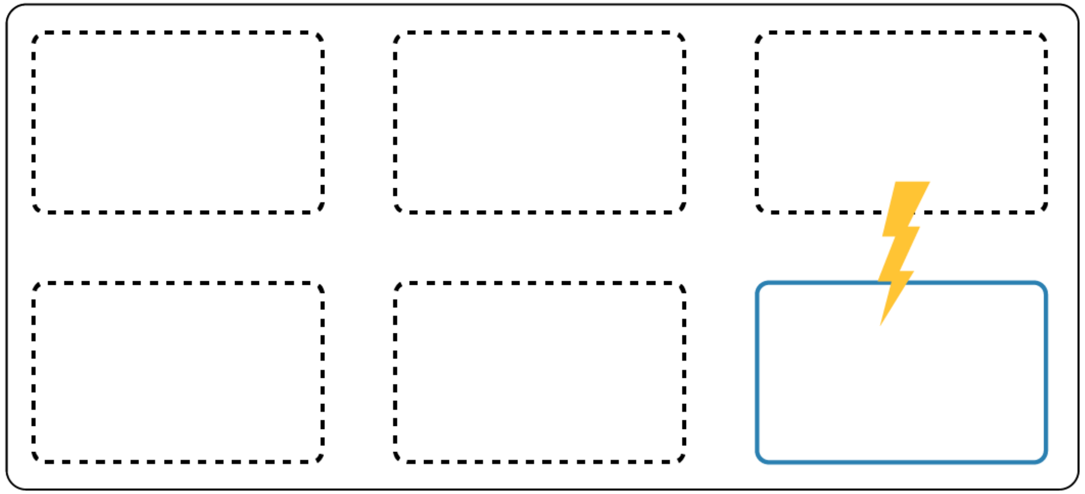
Measurement: Proposed user study (in FY 20) to measure qualities of design and analysis guidance

- Objective qualities
  - Number of issues found / avoided
  - Time required
- Subjective qualities
  - Quality of issues found / avoided
  - Complexity

Using Theory to Guide Tool Development

Approach: Use fault-injection tooling

- Fault-injection pairs naturally with an effects focus
- Collaborators are building a large simulation and verification environment to enable this testing

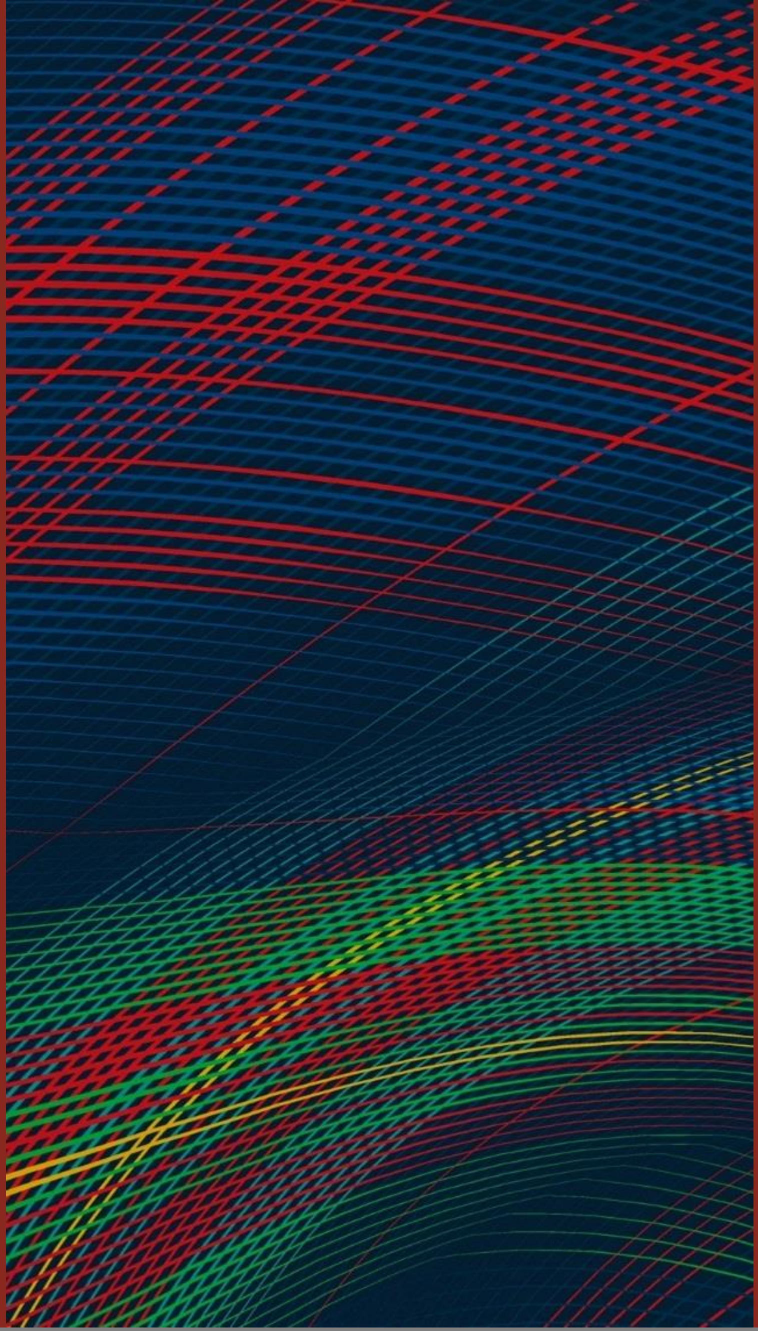


Measurement:

- Current AADL can describe component behavior in the presence of errors
- This project will let us verify those descriptions

Research Review 2018

Using Software Analytics to Analyze Technical Debt



DoD Challenges in Managing Technical Debt

Problem: Government acquirers need capabilities to assess technical debt to manage software schedule and trustworthiness

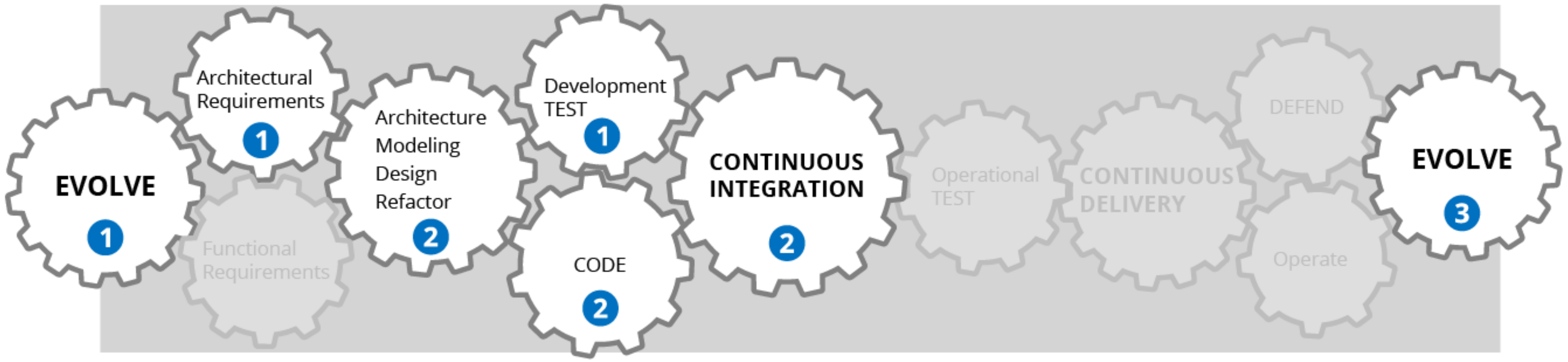
- Differentiate between intentional and unintentional debt
- Prioritize which debt to pay down
- Quantify consequences of technical debt as it remains in the system

Solution: Technical debt analytics will integrate data from multiple sources (code, tickets, code commits) to identify design issues with potential long-term adverse consequences

Automating the Identification of Technical Debt

CMU SEI Approach

1. Build a classifier to detect technical debt discussions in issue trackers
2. Augment static code analysis rules to identify design violations
3. Correlate with commit history data to record candidate technical debt items



Partners

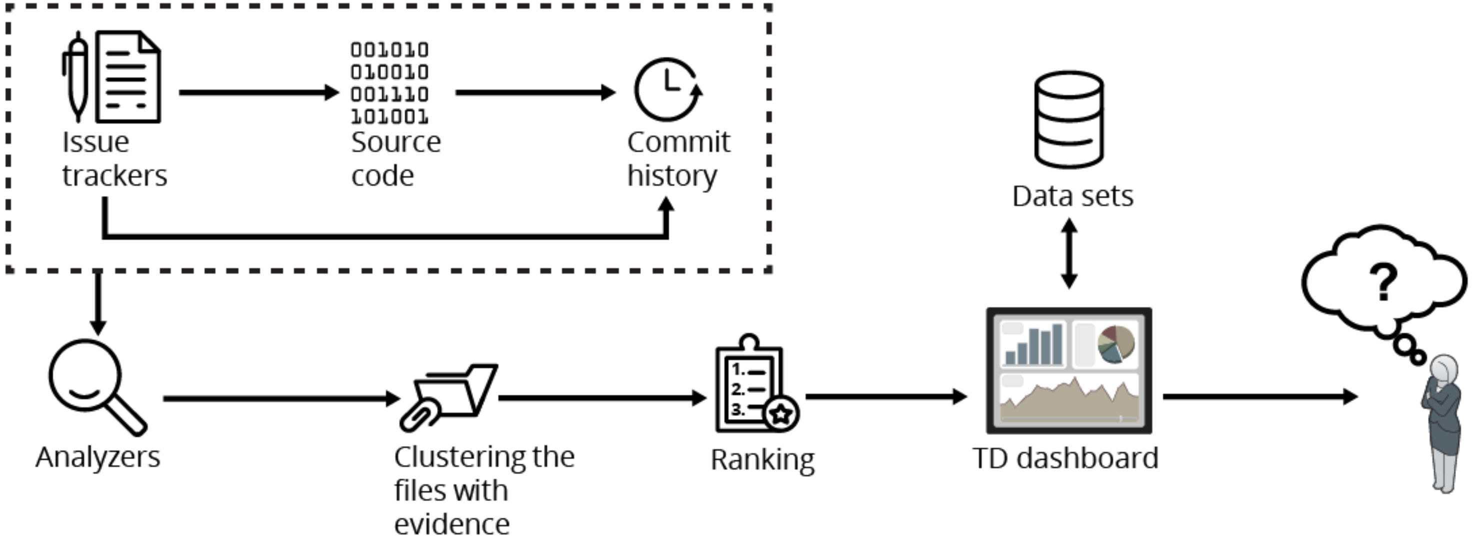
- U.S. Air Force Life Cycle Management Center
- U.S. Food and Drug Administration

Advancing the State of the Art

Developed a TD classifier using machine learning:
Our classifier estimates at least 16% of developer discussions are related to technical debt (data from Chromium open-source issues)

Created design violation analysis augmenting an open-source static code analyzer:
Our algorithm assists in focusing on problematic files, reducing the space of investigation by about 95%

Prioritized candidate technical debt items with supporting evidence
Development teams agree with 80% of the prioritized items as representing technical debt



Detecting Discussions of Technical Debt with Machine Learning

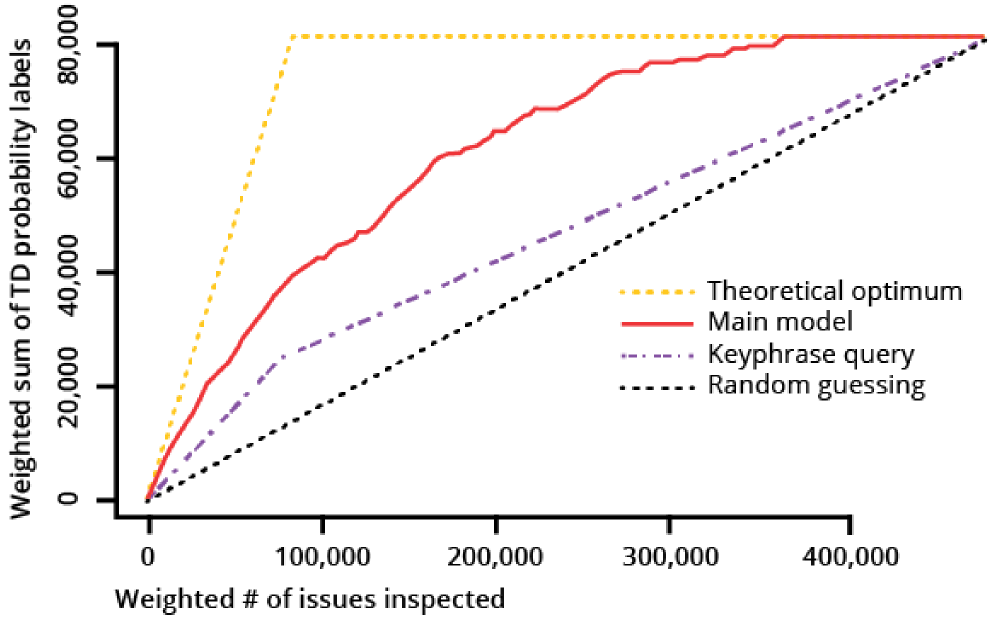
Approach: Uses machine learning, focusing on

- modeling with boosting algorithms to build the weighted average of many classification trees – iteratively improving weak classifiers and creating a final strong classifier
- active learning pipeline and iterating over the data set to use 1,934 labeled technical debt examples
- feature engineering to combine discussion length, *n*-grams, key phrases, concepts, and document context

Using Chromium project with 475,000 issues

Performance metrics

	Accuracy*	Precision*	Recall*	AUROC*
no TD	0.90	NA	0.00	0.50
keyphrase query	0.83	0.26	0.35	0.62
main model	0.87	0.40	0.62	0.88

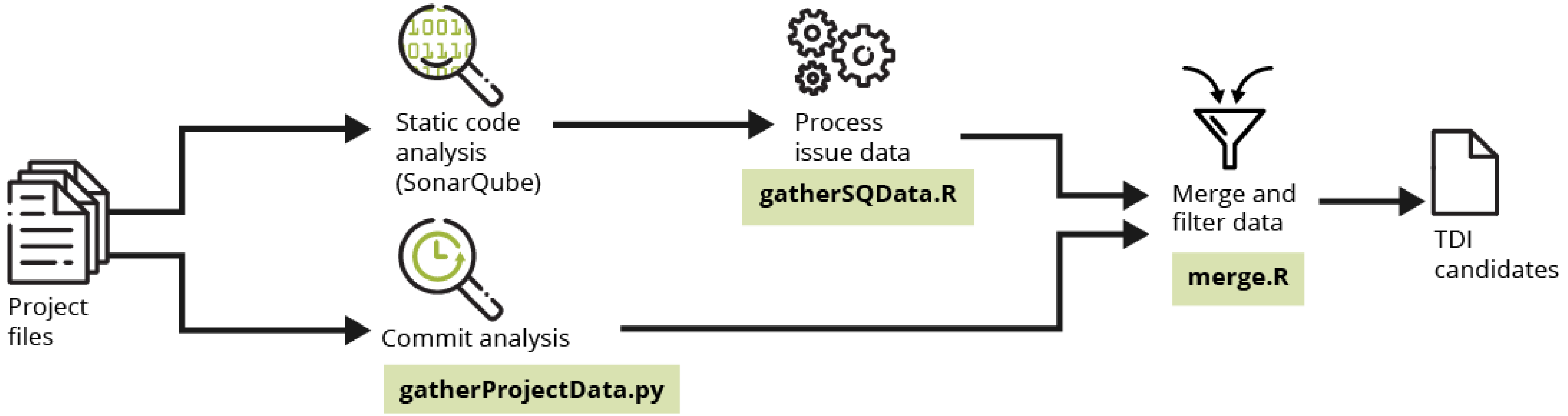


Automating Technical Debt Analysis on Code

Approach:

Combine data from static code analysis and commit history analysis to locate areas of code that hold candidate technical debt items (TDI)

- Augment static code analysis with design rule analysis
- Correlate with commit history profiles of files co-changing and co-committed
- Apply to open-source as well as collaborator data



Developing Analysis Rules and Design Topics

Applied design rule extraction to nine collaborator projects and followed three for longitudinal analysis:

- Teams consistently rate maintainability issues as low priority; in fact, teams change the priority of rules to remove the noise
- The algorithm can identify design problems such as logging, exception handling, and synchronization that should have been acted on earlier

334	Standard outputs should not be used directly to log anything	DR
39	Nested code blocks should not be used	DR
20	@FunctionalInterface annotation should be used to flag Single Abstract Method Interfaces	DR
20	clone should not be overridden	DR
19	Classes should not be too complex	DR
17	**“Exception” should not be caught when not required by called methods**	DR
15	Methods should not have too many parameters	DR
10	Classes named like “Exception” should extend “Exception” or a subclass	DR
3	**Exception handlers should preserve the original exceptions**	DR
2	**Throwable and Error should not be caught**	DR
2	Credentials should not be hard-coded	DR

Example Candidate Technical Debt Items

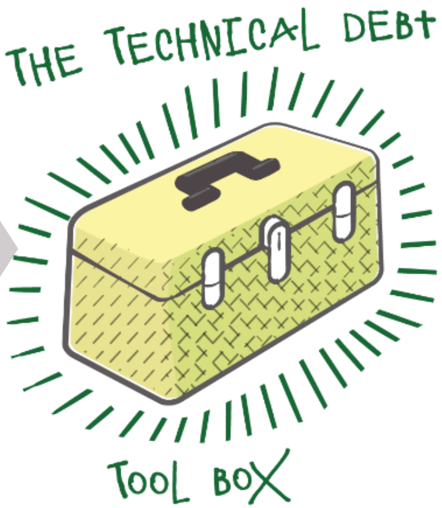
Hadoop: 12,365 files → 61 files; 43 clusters

TDI candidate	Evidence	Design paradigm ➔ Technical debt issue
DFSA*.java**	**Top in DR violations (282)**	**Logging should be centralized to avoid security and other data management issues**
DFS****s.java	DR violations (151)	Credentials and IP addresses should not be hard coded Deprecated code should be removed
F******t.java	DR violations (106)	Redundant exceptions propagate errors and create vulnerabilities
DFS**.java - Dis*****.java**	**Top 2% of files in DR violations High % of commit coupling**	**Redundant exceptions propagate errors and create vulnerabilities and resource management issues Connected files propagate issues**
FS*****.java - F****m.java - F****ry.java - B****.java - F*****p.java	Top 2% of files in DR violations High % of commit coupling	Redundant exceptions propagate errors and create vulnerabilities and resource management issues Connected files propagate issues

Your Technical Debt Toolbox

How does SEI accelerate progress?

- Develop policy guidance
- Develop organizational practices
- Extend and develop tools
- Support data analysis
- Build a community of research and practitioners



What can you and your teams do today?

- Become aware of debt
- Assess the debt
- Build a technical debt registry
- Decide what to fix
- Take action

Looking Ahead:

Increasing Automation to Assist Evolving Systems

FY 19–21: Evolving DoD Software Affordably

Problem: Refactoring is a slow, labor-intensive activity

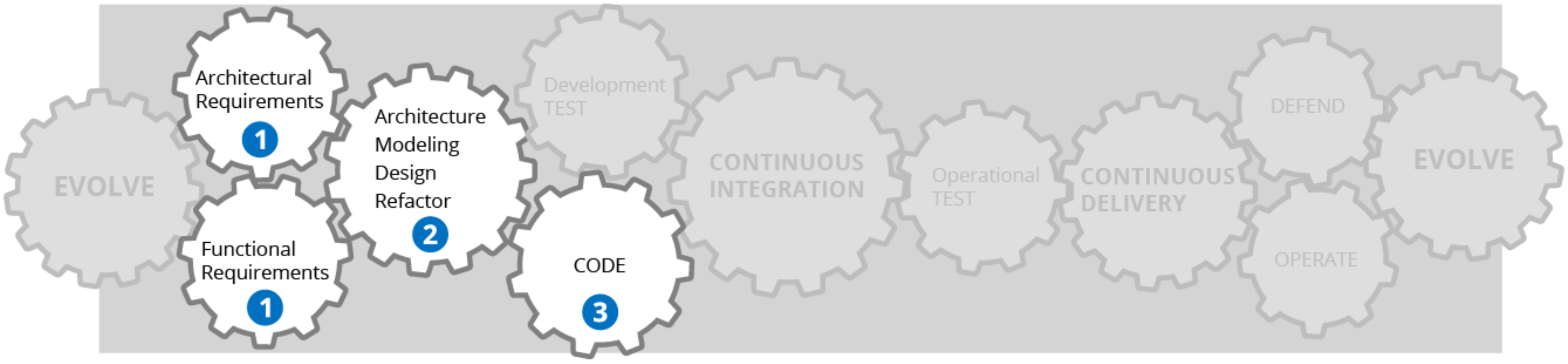
- Harvest components for use in a next-generation system
- Replace a proprietary component
- Reduce coupling with hardware platform

Solution: Create an automated-component refactoring assistant that recommends architectural refactorings and implements them through code transformations

FY 19–21: Evolving DoD Software Affordably

CMU SEI Approach

1. Formalize the evolutionary goal, and use it to drive recommendations
2. Digest and derive existing architecture
3. Adapt search-based algorithms to generate suitable code recommendations



FY 19: Using Machine Learning for Software Development

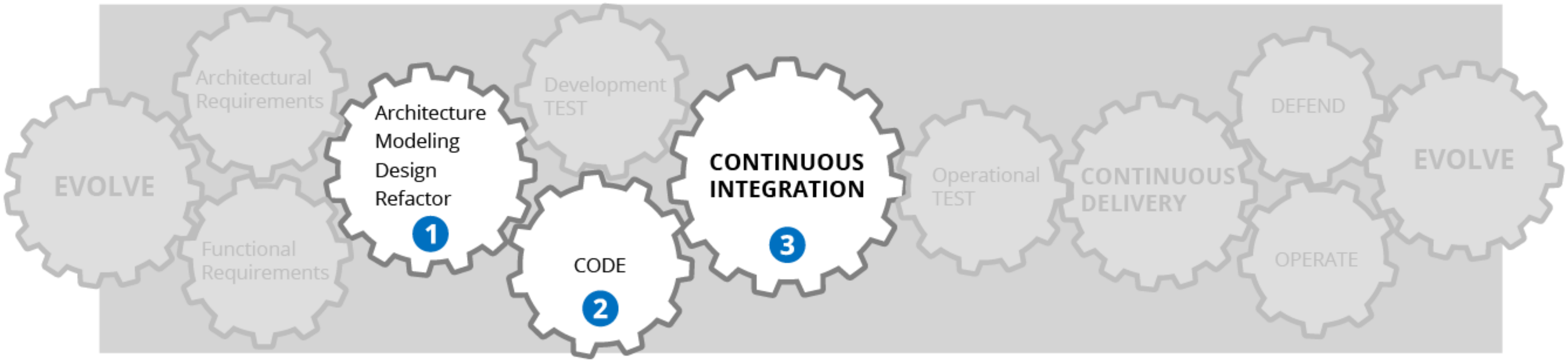
Problem: Inability to detect and enforce use of design patterns limits DoD’s capability to develop affordable and trustworthy software

Solution: Create a proof-of-concept tool using code analysis and deep learning to automatically detect most commonly seen design patterns, in particular model–view–controller

FY 19: Using Machine Learning for Software Development

CMU SEI Approach

1. Bridge abstraction gap between code and design patterns
2. Represent code for machine learning
3. Publish data, and iterate



Long-term vision: integrate into continuous-integration tool chains

[1] Design features on code → [2] Pattern detection → [3] Design enforcement during development

Impact: Advancing the State of Practice

Tooling:

- osate.org
- sei.cmu.edu/go/technicaldebt

Transition partners:

- U.S. Army Joint Multi-Role Tech Demo
- U.S. Air Force Life Cycle Management Center
- U.S. Food and Drug Administration

Community:

- savi.avsi.aero
- techdebtconf.org