Child Welfare Digital Services Project

Test Management Approach

July 2018

Revision History

Revision / Version #	Date of Release	Author	Summary of Changes
0.1	12/06/2016	C. Blehm	Original Document, using the Test Approach document from John Simko as a foundation
0.2	12/09/2016	C. Blehm	Solicit feedback on Test Approach from Ben Hafer's team and incorporate feedback into Alpha version of the document – use John Simko's Test Strategy document as a guideline
0.3	12/14/2016	C. Blehm	Incorporated information from vendor SOW for testing obligations.
1.0	12/29/2016	C. Blehm	Include end-to-end testing approach and strategy or UAT. Identify the process for Defect Management. Revise roles and responsibilities. Define process for sprint testing versus release testing.
1.1	01/21/2017	C. Blehm J. Simko	Make revisions to the UAT approach and insert diagrams for testing approach and roles and responsibilities. Added diagrams for testing approach, testing roles and testing skill sets.
1.2	01/25/2017	C. Blehm K. Bruns	Made revisions to the document per feedback from Agile Coach
1.3	01/31/2017	C. Blehm M. Spolidoro	Made revisions to the document per feedback from PMO Staff
1.4	07/13/18	K. Borini	Incorporated the updates from the WIKI Test Management Approach

Table of Contents

1 Executive Summary

The purpose of the CWDS Test Management Approach is to provide an overall framework and a set of principles, best practices, and guidelines for how testing will occur across all CWDS digital services in delivering code to production frequently as part of the continuous delivery pipeline. This document defines the overall testing strategy, but does not contain a complete set of detailed processes and procedures for each type of test. The primary objective of testing is to validate that the CWS-NS meets the approved user stories and requirements and that it delivers a quality application with minimal defects into production.

The Test Management Approach achieves the following objectives:

- Defines the goals and approach of the CWS-NS Testing processes.
- Defines the test management methodologies, best practices, roles and responsibilities, training and communication required throughout the continuous delivery pipeline.
- Identifies the process for defect management.

Testing is iterative, incremental, and continuous. This rapid turnaround of test results insures the product meets quality standards and facilitates moving the project forward. By providing feedback of test results on an on-going basis, developers are insured the product quality meets the business requirements.

- **Continuous Delivery Pipeline** – The pipeline represents the project's software delivery process for delivering useful, high-quality, working software to users in an efficient, fast, reliable, and repeatable manner. The pipeline breaks down the software delivery process into stages. Each stage is aimed at verifying the quality of the product from a different perspective and prevent identified bugs from affecting users.
- **Test Driven Development** - TDD is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only.
- **Continuous Integration** - CI is a software development process of integrating developers' work frequently (at least daily) using an automated suite of tools. There are three important and foundational elements that underpin any Integration system: (1) Automated Tests; (2) Never break the build; and (3) Version Control.

Various Testing Types include:

1. **Code Quality (development)** – Development teams may use an automated tool to evaluate the code base and ensure compliance with the code-style guideline.
2. **Unit Tests (development)** - tests verify functionality of a small subset of the system, such as an object or method. Automated unit tests are created by the developers to validate the internal function of the software at the service or

component level. They are added to the regression test suite within the CI process as they are developed.

3. **System Tests (development)** - Automated system tests validate an entire module and will be executed automatically as part of the CI process after all unit tests are passed.
4. **Code Coverage (development)** - Code coverage measures how much application code is executed by test scripts. Each digital service team uses automated code coverage tools as part of their CI build process to generate reports that indicated the amount of coverage.
5. **Browser Tests (development)** - Each of the front-end development teams are expected to run automated tests to verify that their components are compatible with various browsers.
6. **Accessibility Tests (development**) - Accessibility testing is a subset of usability testing where the users have disabilities that affect how they use the application. This includes (but is not limited to) ensuring visibility, keyboard navigability, language clarity, and screen reader compatibility.
7. **Smoke Tests (all)** - Smoke tests, also known as “Build Verification Tests”, are comprised of a non-exhaustive set of tests that aim at ensuring that the most important functions work. The results of this type of testing is used to decide if a build is stable enough to proceed with further testing.
8. **Integration tests (Pre-Int Integration**) - Integration tests verify the interaction between code components. They run without the intervention of the GUI web interface.
9. **Acceptance tests (Pre-Int Integration)** - Acceptance tests define the expected behavior for the code to be delivered by the story during the sprint. They are scenarios which are derived from acceptance criteria outlined in the user stories.
10. **Legacy Tests (Integration, Performance, PreProd)** - The legacy systems, specifically CWS/CMS, have many difficult to see “hidden” rules, which requires legacy experts to help identify. Initially, legacy verification will require a lot of people to develop test scenarios and manually execute tests and participate in the acceptance process.
11. **Exploratory Tests (Pre-Int, Preview, Integration, PreProd)** - Exploratory tests are a complimentary activity to automated testing and used verify the quality of the software using a guided methodology that focuses on scenarios that emulates the way a real-world user would use the application.
12. **Usability Testing (Development, Preview)** - Usability tests help evaluate how real users experience the application under realistic conditions. It provides an opportunity to watch real people using the product.
13. **Load and Performance Tests (Performance)** - Load testing evaluates the robustness of the application under higher-than-expected loads. Performance testing identifies bottlenecks in a system.

2 Introduction

In the world of software development, the term agile typically refers to any approach that strives to unite teams around the principles of collaboration, flexibility, simplicity, transparency, and responsiveness to feedback throughout the entire process of developing a new program or product. Agile Testing generally means the practice of testing software for bugs or performance issues within the context of an agile workflow.

Agile suggests that development and testing functions be merged—not necessarily in terms of people, but in terms of time and process—thus bridging the divide between code creators and code breakers, and even reducing the need for robust testing teams, while still respecting the necessity of both roles. In Agile, developers are encouraged to think more like testers, continually checking their own code for potential errors, and testers are encouraged to think more like developers, tempering their natural destructive tendencies to engage more fully in the creative process itself.

> Testing is not a phase in Agile terms. Testing is a way of life. Agile teams must test continuously.

Testing becomes an essential component of each and every phase of the developmental process, with quality being "baked in" to a product at every stage of its development through constant feedback from everyone holding a vision of the final product. Software developers, testers, and quality-assurance personnel wear each other's hats from time to time, and while there may be a select group of people running most of the tests, the notion of a separate testing team disappears entirely for many product teams.

Testing will focus on Fit for use over technical efficiency. Testing will also focus on uncovering bugs over confirming conformance to requirements.

2.1 Purpose

The purpose of the CWDS Test Management Approach is to provide an overall framework and a set of principles, best practices, and guidelines for how testing will occur across all CWDS digital services. This document defines the overall testing strategy, but does not contain a complete set of detailed processes and procedures.

The primary objective of testing is to validate that the CWS-NS meets the approved user stories and requirements and that it delivers a quality application with minimal defects into production.

The Test Management Approach describes the methods that are used to test and evaluate the CWDS system. The approach provides guidance for the management of test activities, including organization, relationships, and responsibilities. The methodology provides a basis for verification of test results of the system. The validation process ensures that the system conforms to the functional requirements and ensures other applications or subsystems are not adversely affected.

2.2 Scope

The Test Management Approach achieves the following objectives:

- Defines the goals and approach of the CWS-NS Testing processes.
- Defines the test management methodologies, best practices, roles and responsibilities, training and communication required throughout the continuous delivery pipeline.
- Identifies the process for defect management.

2.3 Testing Vision

Establish a flexible, scalable, comprehensive, and integrated testing process with the goal of “exactly good enough” using lightweight processes, documentation, tools, and techniques.

2.4 Testing Principles

- All Digital Service team members are responsible for testing and delivering high-quality software
- Quality is built in at the start and tested throughout the life of each Digital Service
- Developers will use the Test-Driven Development (TDD) approach in developing code
- A feature story may not be accepted or meet the Definition of Done until it passes all its acceptance tests
- Bugs identified during testing will be made available to view real-time and on a historical basis
- Provide adequate test coverage to test the software application completely, and make sure it is performing as per the requirements specifications

2.5 Testing Best Practices

- Execute the simplest tests using the simplest tools possible
- Focus on testing the core functionality and “happy path” over edge cases and boundary tests
- Manage test artifacts in source code control repository
    - Map versions of test scripts to the versions of code they test
    - Organize the repository for application code, unit test scripts, and higher-level test scripts
- Set team rules, like the following:
    - Focus on finishing one before starting another
    - The number of open bugs in a sprint should never get higher than 10 at any one time

- Emphasize conversation and collaboration to clarify requirements over specifications and documentation
- Try to have automated tests that can run completely in memory and minimize tests that require database access or have to go through a presentation layer
- Ensure that test coverage remains above the agreed-upon baseline level
- Ensure running times of the automated test suites don't exceed agreed-upon upper bounds
- Conduct an “engineering sprint” or “refactoring iteration” every few months to reduce technical debt, upgrade tool versions, and try out new tools, rather than delivering new functionality/features
- Include time for writing tests, fixing bugs, and preparing test data when planning a sprint
- Whenever a test fails during the continuous integration (CI) build process, the team’s highest priority should be to get the build passing again

2.6 Integration with other CWDS Plans

This plan integrates with the Release Management processes.

Release Management Plan

The Release Management Plan describes the process how the Project will plan, build, monitor and report out progress of CWS-NS releases. The purpose of this plan is to document how the Project will plan, build, test, manage, and govern releases as part of the continuous delivery pipeline.

2.7 Assumptions

Assumptions
• Digital Service RFOs state that they must use Github for Defect Management, however the Digital Service teams are currently using JIRA to log bugs.
• This document is more of a high-level approach and strategy document and not intended to be a typical PM plan with detailed processes and procedures.

2.8 Document Maintenance

The CWDS Test Management Approach will be updated as processes and procedures change. A minor version change does not change the intent of the document and consists of spelling, grammatical and minor corrections. A major version is when a document's content is changed and represents a change in intent, change in process, or procedures. Please refer to the CWDS Configuration Management for further detail on version control.

3 Roles and Responsibilities

Testing is primarily carried out by the digital service development teams, but is also supported by other project roles on the project using a "whole team" approach to testing. Each digital service team's Product Owner is ultimately responsible for ensuring that the appropriate level of quality is being applied and validating that all stories have met the defined story acceptance criteria. The following table describes the roles and responsibilities of the CWS-NS Project stakeholders in the test management arena.

3.1 Testing Roles and Responsibilities Matrix

Table 3 - Testing Roles and Responsibilities Matrix

ROLE	RESPONSIBILITY
Checks and Balances	• Provide independent oversight of testing issues and areas of non-conformance in accordance to CA-PMF and IEEE (where appropriate). • Observe sprint reviews, release planning and execution and testing activities.
Core County Users	• Engage in digital service teams and represent counties from all California regions • Identify performance measurements. • Represent the interests of all counties for that digital service, from the start of work through implementation of that digital service. • Participate in all ongoing development and operations after implementation.
CWDS Quality Assurance	• Provide enterprise quality assurance support services to the state during the agile life cycle. • Ensure compliance with the Project Management Body of Knowledge (PMBOK) guidelines, California Project Management Methodology (CA-PMM), OSI Best Practices (http://www.bestpractices.osi.ca.gov/), industry standards, Information Technology Infrastructure Library (ITIL) standards. • Develop and maintain compliance to the CWDS Quality Management Plan. • Develop and maintain compliance to the CWDS Quality Metrics Plan. • Define the standards to which quality will be measured. • Measure and improve process and product quality.
Lead QA Engineer	• Provide technical leadership, driving and performing engineering best practices to initiate, plan, and execute large-scale, cross-functional, and CWDS-wide critical programs. Lead, build and maintain sets of more advanced (e.g., negative, edge case) feature, integration, and regression tests using industry standard tools. • Lead the creation, modification, execution and maintenance of feature, integration, and load/performance test scripts along with full regression tests. • Lead the building and maintenance of a library of reusable scripts and processes.

ROLE	RESPONSIBILITY
	• Log and track software defects. • Become the key source of institutional knowledge (across multiple/changing product owners) of how the system as a whole works. • Lead the building and maintenance of sets of integrated Child Welfare Services – New System (CWS-NS) and Child Welfare Services / Case Management System (CWS/CMS) tests to validate CWS/CMS. • Lead the effort to closely work with Digital Service, Technical Platform, and DevOps teams through the entire project lifecycle to ensure overall software system quality is maintained. • Lead the gathering and analysis of requirements to define, implement, and maintain test plans, test specifications and test suites. • Provide child welfare services business subject matter expertise. • Work closely with Legacy Testing team to understand requirements and translate them into test cases. • Provide feedback to Digital Service, Technical Platform, and DevOps teams on software usability and functionality. • Develop strong working knowledge of child welfare services business practices. • Participate in the development of project-approval documents such as the state Special Project Report (SPR) and federal Advanced Planning Documents (APD). • Prepare various sections of the SPR and APDs in accordance with guidelines and regulations stipulated in the Statewide Information Management Manual and the Code of Federal Regulations respectively. • Assist in responding to control agency observations and coordinate responses from CWDS and CDSS. • Also, assist in the development of ad hoc reports for the Legislature, Administration and other entities.
Digital Service Development Teams	• Work with the Product Owner to articulate the user story requirements into executable acceptance tests. • Define testing acceptance criteria to conduct unit, functional, story acceptance, integration, and performance testing. • Develop and execute automated unit, functional, story acceptance, integration, and performance tests. • Respond to quality review and audit findings as part of the quality improvement process. • Use an automated tool for static code analysis to evaluate the codebase and ensure compliance with the code-style guidelines. • Use an automated tool that measures the amount of the codebase that is covered by tests. • Create and execute automated integration testing with other contractor developed modules.
Legacy Testers	• Provide child welfare services business subject matter expertise. • Build and maintain sets of integrated CWS-NS and CWS/CMS tests to validate CWS/CMS. • Perform regression tests against CWS/CMS to ensure functionality is not broken due to CWS-NS changes.

ROLE	RESPONSIBILITY
	• Work closely with Digital Service and Tech Platform teams through the entire project lifecycle to ensure overall software system quality is maintained. • Work closely with QA Engineering team to develop test scripts. • Gather/analyze requirements and develop test plans on new/existing features. • Do manual testing in the early phases of product development. • Create, modify, execute and maintain feature, and integration test scripts along with full regression tests. • Build and maintain a library of reusable scripts and processes. • Provide feedback to Digital Service, Tech Platform teams on software usability and functionality • Log and track software defects.
Product Owner or Service Manager	• Determines the Sprint Definition of Done • Participate in Bug triage activities • Prioritize Bug Stories • Approves user story acceptance criteria • Participate in service team sprint reviews to identify testing issues or concerns. • Escalates testing issues or risks to the Risk manager as needed. • Communicate testing issues and risks to internal and external stakeholders.
Project Director	• Communicate testing issues and risks to internal and external stakeholders. • Communicate with ELT to report any testing related issues.
QA Engineer	• Build and maintain sets of more advanced (e.g., negative, edge case) feature tests, integration tests, regression tests and load/performance tests. • Work closely with Digital Service, Tech Platform, and DevOps teams through the entire project lifecycle to ensure overall software system quality is maintained. • Work across digital service teams to ensure integration across all services. • Work closely with Legacy Testing team to develop test scripts. • Gather/analyze requirements and develop test plans on new/existing features. • Do manual testing in the early phases of product development if automated tests are not ready. • Create, modify, execute and maintain feature, integration, and load/performance test scripts along with full regression tests. • Build and maintain a library of reusable scripts and processes. • Provide feedback to digital service, Tech Platform, and DevOps teams on software usability and functionality • Develop strong working knowledge of child welfare services business practices. • Log and track software defects. • Become a key source of institutional knowledge of how the system as a whole works.
Scrum Master	• Collect and validate testing metrics and/or data analytics from Dev team • Manage and track bug stories
Security Tester	• Configure and manage automated black/white box security scanning tools. • Develop data driven automated security tests scripts. • Identify infrastructure and data required to conduct security testing. • Execute security tests and collate the results.

ROLE	RESPONSIBILITY
	• Identify, mitigate, communicate out, and escalate defects, issues, and risks related to security testing. • Oversee the digital services vendors on security testing processes, review vendor developed security testing scripts, evaluate deliverables for acceptance, and resolve security testing questions throughout the project life cycle. • Provide project staff with training and an understanding of security testing tools, techniques, and procedures. • Research and evaluate automated security scanning tools. • Monitors evolution of industry security testing best practices and standards during CWS-NS maintenance and operations phase of the system lifecycle to identify new security testing requirements.
Technology Manager	• Establish and execute technical policies, processes, and procedures and ensure adherence to defined testing standards

4 Test Management Approach

This section describes the approach for supporting testing and addresses the various types of testing and quality assurance activities, that occurs throughout the continuous delivery pipeline in getting releases out the door and into user's hands. Testing is iterative, incremental, and continuous. This means that each increment of coding is tested as soon as it is finished. The team builds and tests a little bit of code, making sure it works correctly, and then moves on to next piece that needs to be built. This rapid turnaround of test results insures the product meets quality standards and facilitates moving the project forward. By providing feedback of test results on an on-going basis, developers are insured the product quality meets the business requirements.

4.1 Continuous Delivery Pipeline

As described in the CWDS Release Management Plan, the Continuous Delivery (CD) Pipeline (hereinafter referred to as the "pipeline") represents the project's software delivery process for delivering useful, high-quality, working software to users in an efficient, fast, reliable, and repeatable manner. The pipeline breaks down the software delivery process into stages. Each stage is aimed at verifying the quality of the product from a different perspective and prevent identified bugs from affecting users. Releases are not allowed to move forward to the next stage unless they pass all Stage Acceptance Criteria or have an approved exception.

Figure 1 – Logical Continuous Delivery Pipeline describes the logical migration path that a release candidate will take from one stage to the next on its journey to production (aka Live).

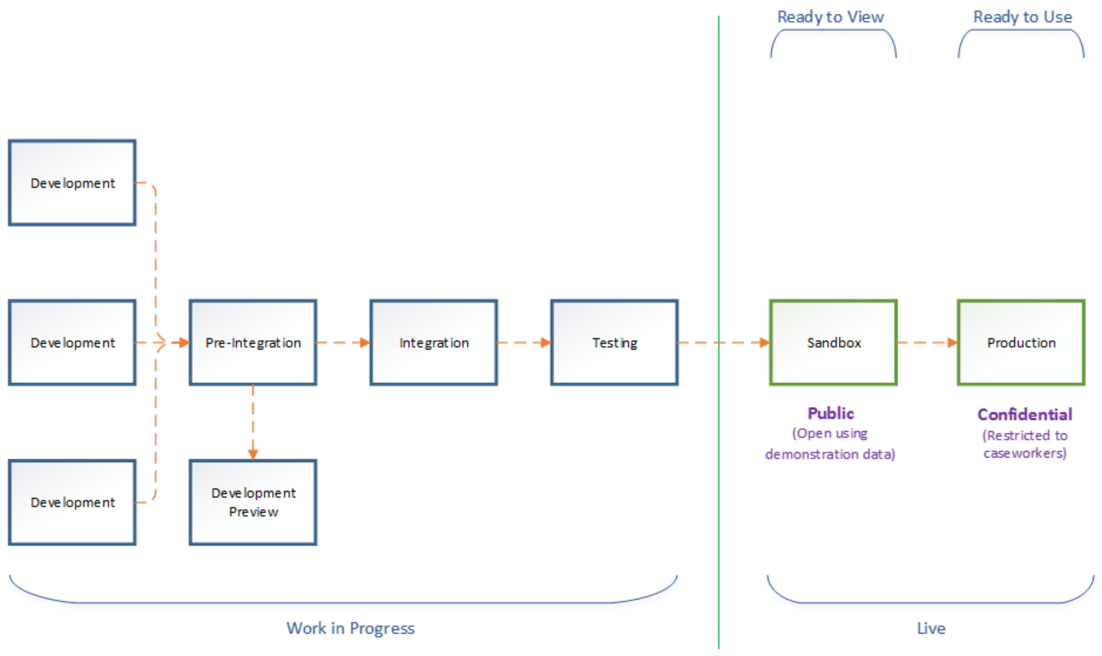
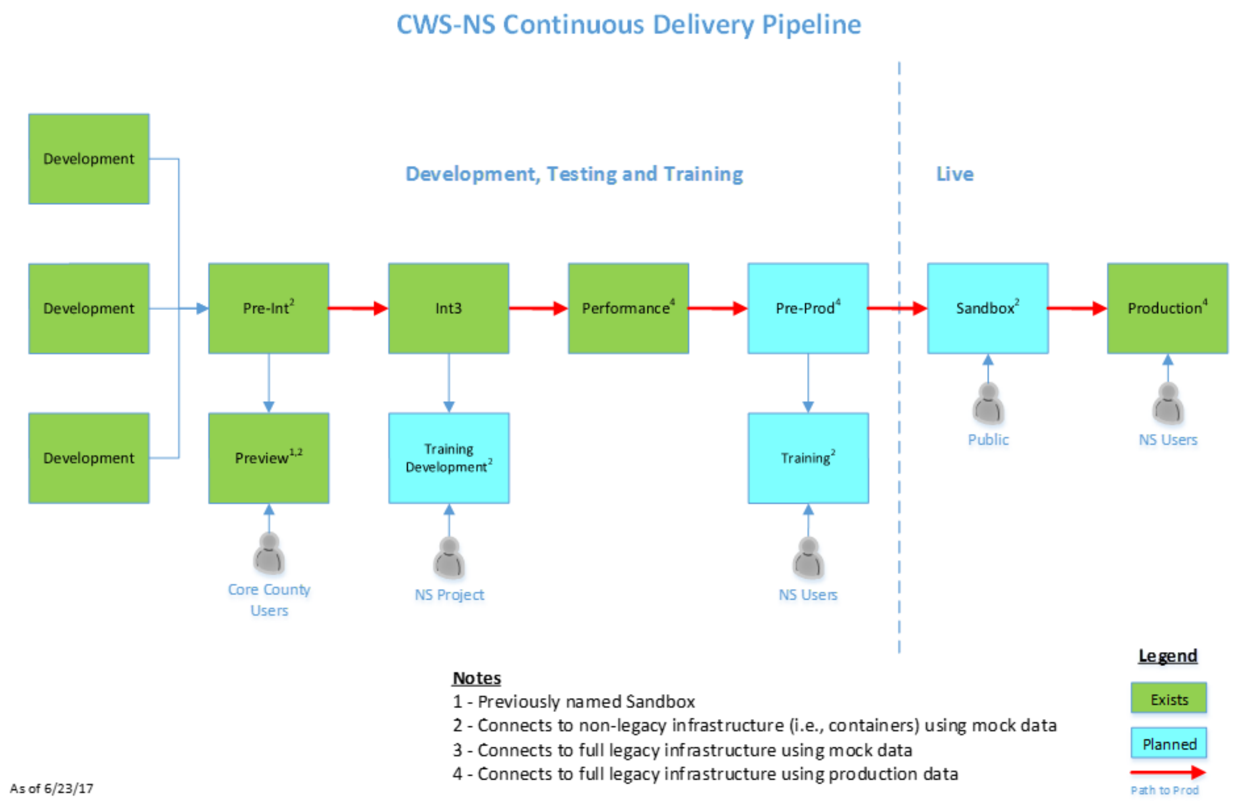


Figure 2 – Physical Continuous Delivery Pipeline

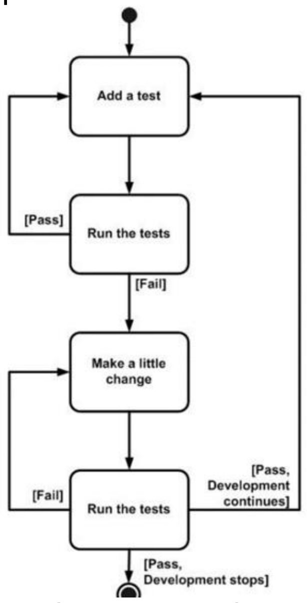
Figure 2 – Physical Continuous Delivery Pipeline describes the actual migration path that code will take from one stage (environment) to the next on its journey to production. The pipeline starts in the Development stage/environment with each development team building, testing and deploying code following a defined continuous integration (CI) process. At each subsequent stage in the CD pipeline, the application is tested to ensure that it meets all desired system qualities appropriate for the environment/stage that it is in. Only once the application passes each stage can the feature be released to production.


CWS-NS Continuous Delivery Pipeline
Development, Testing and Training
Live
Development
Development
Development
Pre-Int2
Int3
Performance4
Pre-Prod4
Sandbox2
Production4
Preview1,2
Training Development2
Training2
Core County Users
NS Project
NS Users
Public
NS Users
Notes
1 - Previously named Sandbox
2 - Connects to non-legacy infrastructure (i.e., containers) using mock data
3 - Connects to full legacy infrastructure using mock data
4 - Connects to full legacy infrastructure using production data
Legend
Exists
Planned
Path to Prod
As of 6/23/17

4.2 Test Driven Development

During the Development/Build stage, each of the development teams are utilizing a software development process called Test Driven Development (TDD). TDD is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. TDD uses the following steps:

- Write the test
- Run the test (there is no application code, test does not pass)
- Write just enough application code to make the test pass
- Run all tests (tests pass)
- Refactor
- Repeat



Step 1: Write the Test. In test-driven development, each new feature begins with writing a test to define a function or improvements of a function. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference

Step 2: Run the test and see if it fails. This validates that the test harness is working correctly. A test harness is a collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs. Step 2 also shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

Step 3: Write the code. The next step is to write some code that causes the test to pass. The new code written at this stage may not be efficient or elegant, but the test is passed." That is acceptable because it will be improved and honed in Step 5.

Step 4: Run tests. If all test cases now pass, the developer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

Step 5: Refactor Code. The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. However, duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and for creating clean code. By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to the removal of any duplication between the test code and the production code—for example magic numbers or strings repeated in both to make the test pass in Step 3.

Step 6: Repeat. Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

4.3 Continuous Integration (CI)

A key component of the pipeline is Continuous Integration (CI), which is a software development process of integrating developers' work frequently (at least daily) using an automated suite of tools.

There are three important and foundational elements that underpin any Continuous Integration system.

1. **Automated Tests.** The commitment of the development team to produce a comprehensive automated test suite at the unit level and functional level together with their code. It is essential that the automated tests are run with each build to identify potential bugs in a timely manner.
2. **Never break the build.** The goal is that the application must be ready to be built, packaged and deployed on every committed change. This means that broken or untested code should never be committed.
3. **Version Control.** The code has to be managed by strict version control policies which includes the practices of frequent commits to a version control repository, fixing broken builds immediately, and using a separate integration build machine.

4.4 Pair Programming

Another strategy for ensuring high-quality code is the approach of pair programming, which entails two programmers working together to complete a set of code. One, the driver, writes code while the other, the observer or navigator, reviews each line of code as it is typed in. The two programmers switch roles frequently. Each of the development teams are expected to implement some form of this in meeting their team's sprint definition of done. The following is an example of the pair programming process that the Intake team is following

- A pair of Developers usually pick up story from top of JIRA backlog.
- Assume all stories are prioritized and estimated in JIRA backlog
- Pair starts the story in the JIRA
- Pair creates feature branch off latest master branch
- Completes story using Test Driven Development
- Submits pull request to Tech lead and peer developers
- Pull request is approved after code review by lead or peer developers
- Merge changes to master branch
- Pair finishes story in JIRA
- Continuous Integration runs automated tests and deploys changes to acceptance environment
- If Continuous Integration is green, then pair delivers story in JIRA

4.5 Quality Acceptance

4.5.1 Story Acceptance Criteria

Acceptance Criteria is the minimal amount of documentation needed to specify the expected behavior of each story/feature.

- Create prior to the start of development
- Produced and validated through collaboration of the developers, testers, and product owners
- Helps the team gain a shared understanding of the story/feature
- Defines the boundaries for a user story/feature
- Helps developers and testers to derive tests

Quality is maintained continually with the agile development approach. Both the testers and developers have more confidence in the code: they know there is a reduced chance that undiscovered defects will create problems downstream.

4.5.2 Continuous Delivery (CD) Stage Acceptance Criteria

CD Acceptance Criteria specifies the criteria that each software candidate release must meet in order to exit one of the CD stages and be promoted to the next stage. The criteria is different for each stage based on the types of testing that occurs in the stage.

Details of each stage’s acceptance criteria can be found in the CWDS Release Management Plan.

4.5.3 Definition of Done

Each Scrum Team, along with the Product Owner and Scrum Master, commonly defines the “Definition of Done” to claim that something is completed (story, sprint, release). Usually it includes incremental product architectural design, feature design, development, unit testing, integration testing, functional testing, and documentation.

The Statement of Work (SOW) for each development team includes a requirement to define the sprint-level Definition of Done that includes the following concepts:

- Code produced (all ‘to do’ items in code completed)
- Code commented, checked in and run against current version in source control
- Peer reviewed (or produced with pair programming) and meeting development standards
- Builds without errors
- Unit tests written and passing
- Deployed to system test environment and passed system tests
- Passed Service Manager Acceptance Testing
- Any build/deployment/configuration changes Implemented/documented/ communicated
- Relevant documentation produced/updated (e.g., how the API services supports service module needs and user stories, user stories, sketches, wireframes, clickable prototypes)
- Remaining hours for task set to zero and task closed

4.5.4 Automated Testing

All team members, in particular, developers should be involved in creating and maintaining automated tests, and should stop what they are doing and fix them whenever there is a failure. Furthermore, each of the digital service vendors are expected to implement as much automated testing as possible. The Statement of Work (SOW) for each development team includes the following requirements for automated testing:

- The Contractor shall create and execute automated unit testing
- The Contractor shall create and execute automated system tests to verify all Features of the software module
- The Contractor shall create and execute automated Service Manager Acceptance testing to verify all user facing functionality
- The Contractor shall run tests automatically on code merged into version control
- The Contractor shall create and execute automated integration testing with other contractor developed modules
- The Contractor shall ensure that all code will be written to a language specific code-style guideline (e.g., PEP8 for Python)

- The Contractor shall use an automated tool to evaluate the codebase and ensure compliance with the code-style guideline (e.g., if the Contractor uses Python, PyLint may be used)
- The Contractor shall use an automated tool that measures the amount of the codebase that is covered by tests (e.g., RCov may be used to measure test coverage of Ruby code)

4.6 Bug Tracking

Currently, all bugs are entered into each team’s JIRA as a Bug story. The bug story should have the following information:

- **Date Detected**: MM/DD/YYYY
- **Description**: Explain the defect, what screen(s) the defect is found on.
- **Steps**: Describe steps taken to recreate the defect.
- **Environment**: e.g., Pre-Int, Integration, Performance
- **Screenshots**: If appropriate, attach screenshots
- **Expectations**: If known, enter the expected behavior

While it is up to each team to determine the triage process that works best for their team, every bug story must be assigned a one of the following JIRA labels that classifies its severity:

- **Bug - Critical**: Severely limits the users' ability to perform their job. Must be fixed before releasing into Production.
- **Bug - Moderate**: limits the users' ability to perform their job. Possible workaround available. May need escalation to determine whether acceptable to release system into production without this bug being fixed. Timeline for fix identified.
- **Bug - Minor**: does not impact user's ability to do their job. Examples: typos, cosmetic errors, text misalignment, etc.

The Product Owner treats each bug like any other story in their backlog and determines its priority.

4.7 Test Reporting

Test reporting supports decision making and provides visibility to the project. The following are examples of the types of metrics that are reported in the Sprint Review Demonstrations:

- # of Bugs incurred in the sprint
- # of open bugs at the end of the sprint
- # of Bugs resolved in the sprint
- # of unit tests run each day
- # of test written, run, and passed at each level (e.g., unit, functional, story, load)
- Code coverage
- Build duration

Each team maintain a dashboard for reporting all major test components and defining a threshold for each component within each digital service team.

4.8 Testing Tools

The CWDS front-end digital service teams (e.g., Intake) use the following tools to assist in testing and code analysis:

- RSpec for Behavior Driven Development improving on Test Driven Development and Unit Testing.
- RSpec with Selenium and Capybara for Integration testing (Behavior Driven Development) and Acceptance Testing
- Selenium with Capybara using RSpec for performance testing
- CodeClimate along with Code Review to Check Code Quality
- Jasmine for Test Driven Development and Unit testing JavaScript Code
- Karma for executing JavaScript Code in multiple real browsers and Test Drive JavaScript Code.
- RuboCop as static code analyzer that enforce guidelines outlined in community Ruby Style Guide
- ESLint for static analyzing JavaScript Code to find problematic patterns or code that doesn't adhere to certain style guidelines
- Capybara-accessible is being used for accessibility tests along with our Rspec integration tests which use Capybara, helping you to capture existing failures and prevent future regressions.
- simplecov is a code coverage tool used to analyze Ruby code.
- istanbul is a tool used to report on the level of javascript code coverage.

The CWDS back-end digital service teams (TPT.1) use the following tools to assist in testing and code analysis:

- SonarQube - Quality Check
- JMeter – Unit, Functional and Performance Testing
- JUnit – Unit Testing
- Cobertura – Code Test Coverage
- Unified Functional Testing (UFT) – Automated Legacy (CWS/CMS) Testing

4.9 Test Data

In the early sprint cycles, dependencies between components that are still under development may require mocking up of test data. Later on, in order to perform certain tests, such as performance testing, production user IDs or data may be needed. This section will identify when test data may need to be created and who is responsible for creating it.

The legacy CWS/CMS system maintains a predefined set of test data that will be leveraged. In addition, each digital service team may come up with their own set of test data for specific test scripts that they develop.

5 Testing Types

The types of testing that will support the CWDS digital service development can be broken out into different types of testing that accomplish different purposes. Next to each type of test listed below is the environment (e.g., Development, Pre-Int) where this testing is typically conducted.

5.1 Code Quality (Development)

In addition to following The Contractor shall use an automated tool to evaluate the codebase and ensure compliance with the code-style guideline (e.g., if the Contractor uses Python, PyLint may be used).

5.2 Unit Tests (Development)

Unit tests verify functionality of a small subset of the system, such as an object or method. Automated unit tests are created by the developers to validate the internal function of the software at the service or component level. They are added to the regression test suite within the CI process as they are developed. Each time new code is checked in, the CI executes the total suite of automated unit tests (regression tests), ensuring that any code that breaks the integration will be caught early in the process. The CI process is usually configured to send automated notifications as build processes are executed – especially failures, but any outcome and/or previous state can trigger a notification (Pass, Fail, Breaks, All, etc.).

5.3 System Tests (Development)

Automated system tests validate an entire module and will be executed automatically as part of the CI process after all unit tests are passed.

5.4 Code Coverage (Development)

Code coverage measures how much application code is executed by test scripts. Each digital service team uses automated code coverage tools as part of their CI build process to generate reports that indicated the amount of coverage. While the overall goal is to strive for high unit test coverage, 100% is not an expected outcome. The teams are expected to focus on tests in critical areas such as business logic and worry less about tests around supporting code such as Plain Old Java Objects (POJOs) for java-based code.

As an example, Intake's Ruby code coverage can be found by navigating to the Intake CI and clicking on 'Ruby Code Coverage'. Their javascript code coverage can be found by selecting the 'JS Code Coverage'.

5.5 Browser Tests (Development)

Each of the front-end development teams are expected to run automated tests to verify that their components are compatible on the following browsers:

- Browser Priority 1 (Standards compliant)
    - WebKit
        - Chrome
        - Safari
        - Edge
    - Gecko
        - Firefox
- Browser Priority 2 (Non-Standards compliant)
    - Microsoft (Legacy)
        - IE v11

5.6 Accessibility Tests (Development)

Accessibility testing is a subset of usability testing where the users have disabilities that affect how they use the application. Accessibility testing will be done throughout the design and development process ensure the system meets Section 508 of the Rehabilitation Act's most recent guidelines (which as of Jan 18, 2017 incorporates WCAG 2.0 guidelines). This includes (but is not limited to) ensuring visibility, keyboard navigability, language clarity, and screen reader compatibility.

As stated in the project's Research-Design wiki page, the project will apply the following design strategy:

- All components of the CWDS Pattern Library meet accessibility requirements. Refer to the 18F Accessibility Guide Checklist.
- Components combined together into page designs are tested holistically to ensure accessibility is maintained.
- Regular usability testing is conducted with disabled users.
- Any gaps in usability will be identified and corrected, or tagged with "accessibility-needs" in our product backlogs and prioritized.
- Periodic review and recommendations by CA Dept of Rehabilitation (DOR).

In addition, the front-end development teams will use an automated accessibility testing tool (Pa11y) to validate compliance as part of their CI build process.

5.7 Smoke Tests (All)

Smoke tests, also known as "Build Verification Tests", are comprised of a non-exhaustive set of tests that aim at ensuring that the most important functions work. The results of this type of testing is used to decide if a build is stable enough to proceed with further testing. Automated smoke tests will be included with the deployment of new builds to each of the stages/environments.

5.8 Integration Tests (Pre-Int and Integration)

Integration tests verify the interaction between code components. They run without the intervention of the GUI web interface.

5.9 Acceptance Tests (Pre-Int and Integration)

Acceptance tests define the expected behavior for the code to be delivered by the story during the sprint. They are scenarios which are derived from acceptance criteria outlined in the user stories. Each acceptance criteria can have one or more acceptance tests. There must be at least one test case created for 100% of the approved acceptance criteria. Testers will also test for a negative and positive outcome for each acceptance criteria. All test cases and subsequent test scripts and expected results are directly related to at least one acceptance criteria for each user story. Acceptance testing best practices include the following:

- Becomes “living” documentation about how a piece of functionality works
- Steps for developing acceptance tests include the following:
    - Write high-level test cases with examples of desired and undesired behavior before coding
        - Start discussions about features and stories with a realistic example
        - Identify which features are critical and which ones can wait
        - Start with happy path
        - Identify all of the parts of the system that might be affected by a story
        - Review test cases with customers
    - Sit down with developers to review the high priority stories and their associated test cases
    - As soon as testable chunks of code are available and the automated tests pass, explore the functionality more deeply by trying different scenarios and learning more about the code’s behavior. The story is not “done until all these types of tests are complete
    - Write detailed test cases once coding starts
- Additional acceptance testing best practices include the following:
    - Identify high-risk areas and make sure that code is written to solidify those risk areas
    - Ensure that the most obvious use-case is working first before tackling challenging areas
    - Understand the domain and the impact of each story. Don’t forget the big picture.
    - Confine each test to one business rule or condition
    - If a developer has started working on a story, there should be a tester working on the testing tasks for that story
    - Time-box the time spent writing tests

5.9.1 Legacy Tests (Integration, Performance, PreProd)

The project's strategy for implementing CWS-NS is based on making legacy parity a priority. This essentially means that each digital service team should build CWS-NS functionality that, at a minimum, doesn't break existing legacy business functionality, while at the same time provides new and improved features and functionality to the users. While legacy testing is not considered its own type of testing, it is important to call it out separately given the complexity and significance of validating the integration of the CWS-NS with the legacy systems. The legacy systems, specifically CWS/CMS, have many difficult to see "hidden" rules, which requires legacy experts to help identify. Initially, legacy verification will require a lot of people to develop test scenarios and manually execute tests and participate in the acceptance process. As we successfully pass our manual tests, we continue to automate as much as possible.

For more details on TPT1 team's approach to legacy development and testing go here and here.

5.10 Exploratory Tests (Pre-Int, Preview, Integration, PreProd)

Exploratory tests are a complimentary activity to automated testing and used verify the quality of the software using a guided methodology that focuses on scenarios that emulates the way a real-world user would use the application. They are manual in nature and the tester uses his or her domain knowledge and testing experience to predict where and under what conditions the system might behave unexpectedly.

- Not a means of evaluating the software through exhaustive testing
- Reveals areas of the product that could use more automated tests
- Identifies ideas for new or modified features that lead to new stories
- Performed as the product is evolving or as a final check before the software is released

5.11 Usability Testing (Development, Preview)

Usability tests help evaluate how real users experience the application under realistic conditions. It provides an opportunity to watch real people using the product. It can give you important metrics such as how long it takes someone to complete a task, or how many clicks it takes to find something. It is not looking for what passes or fails, just looking for how the users are using the functionality. These tests done with each digital service team's core county users and are coordinated by the team's user research and design resources. Usability testing techniques that the digital service teams are using testing include:

- Task-based testing: users are asked to perform a list of tasks and narrate how they do them
- Scenario-based testing: users are provided with a specific scenario and asked to describe how they would handle it

Task-based tests are good for the following:

- Spotting specific points where the usability of a solution breaks down
- Observing problems that were not expected
- Observing problems the user would did not verbalize

Scenario-based tests are good for the following:

- Getting feedback on something that is built
- Identifying problems with a design
- Prioritizing what problems and scenarios to address first

5.12 End User System Feedback (Preview, Demo-Integration, PreProd, and Sandbox)

Unlike the traditional waterfall methodology where User Acceptance Testing (UAT) is traditionally conducted as the last step in the life-cycle development process prior to releasing functionality to production, the project has established a number of different environments as part of its continuous delivery pipeline that will provide opportunities for users to provide feedback. Table 1 – User Feedback Environments describes the various environments that users will interact with in order to provide feedback to the project.

Environment	Preview	Demo-Integration	PreProd	Sandbox
Purpose	Provide project staff as well as core county and state users early access to code and the opportunity to provide feedback	Provide project staff the ability to showcase digital service features to core county and state users as well as other stakeholders	Provide project staff as well as core county and state users the opportunity to validate release features in a production-like environment and provide feedback	Provide project staff, county and state users, stakeholders, and the general public the opportunity to use the system and provide feedback
Participants	• Project staff • Core county and state users	• Project staff (facilitates) • All county and state users • General public	• Project staff • Core county and state users	• Project staff • All county and state users • General public
Types of end user testing	• Non-scripted feature validation • Ad-hoc	• Scripted scenarios	• Non-scripted feature validation • Legacy regression • Network latency • Ad-hoc	• Ad-hoc
Location	Any CWDS location or via VPN	Any location via VPN	Any CWDS location or via VPN	Any location
Data	Mock data that does not contain any personally identifiable information (PII) or protected health information (PHI) data	Mock data that does not contain any personally identifiable information (PII) or protected health information (PHI) data	A copy of legacy production data	Mock data that does not contain any personally identifiable information (PII) or protected health information (PHI) data

Availability	24x7, except for scheduled maintenance	24x7, except for scheduled maintenance	24x7, except for scheduled maintenance	24x7, except for scheduled maintenance
Frequency	New features are added to Preview at the end of every iteration	As required	Prior to each release to production	New features are added to Sandbox at the end of every iteration
Duration	Ongoing	Varies by demonstration	Varies by release, but will be time-boxed (e.g., 2-3 days)	Ongoing
Outcomes	• Identification of new bugs • Identification of new features • Identification of functionality that could use more automated tests	• Identification of new bugs • Identification of new features	• Identification of new bugs • Identification of new features • Identification of functionality that could use more automated tests	• Identification of new bugs • Identification of new features • Identification of functionality that could use more automated tests
Target Environment Build Date	• 10/15/17	• Completed	• 11/30/17	• 11/15/17
Configuration	• User can access using any internet capable device • No access to CWS/CMS client • No interface (e.g., Address validation) capability • Users are added to a LDAP directory • VPN not required	• User can access using any internet capable device • Access to CWS/CMS client • Limited interface (e.g., Address validation) capability • Users are added to SAF and must have a valid legacy (e.g., CWS/CMS) test RACF Id • Requires VPN	• User can access using any internet capable device • Access to CWS/CMS client • Limited interface (e.g., Address validation) capability • Users are added to SAF and also must have a valid legacy (e.g., CWS/CMS) production RACF Id • VPN not required	• User can access using any internet capable device • No access to CWS/CMS client • No interface (e.g., Address validation) capability • No SAF authentication required • VPN not required
Feedback	**Core county and state users:** Feedback is submitted by to	Feedback is captured during the demonstration session	**Core county and state users:** Feedback is submitted by to email	**Core county and state users:** Feedback is submitted by to email

	email addresses set up by each digital service Feedback is triaged by the development teams Incidents are logged as bugs in the appropriate digital service team's icebox Enhancement requests are captured as new feature stories in the appropriate digital service team's icebox and looked at as part of each digital service team's story grooming process	Incidents are logged as bugs in the appropriate digital service team's icebox Enhancement requests are captured as new feature stories in the appropriate digital service team's icebox and looked at as part of each digital service team's story grooming process	addresses set up by each digital service Feedback is triaged by the development teams Incidents are logged as bugs in the appropriate digital service team's icebox Enhancement requests are captured as new feature stories in the appropriate digital service team's icebox and looked at as part of each digital service team's story grooming process	addresses set up by each digital service **General Public:** Feedback will be entered on a feedback web page and stored in a publicly visible page located on the Sandbox GitHub Issues page Feedback is triaged by the project and assigned to the appropriate development team Incidents are logged as bugs in the appropriate digital service team's icebox Enhancement requests are captured as new feature stories in the appropriate digital service team's icebox and looked at as part of each digital service team's story grooming process.
Release Notes	Published in the Preview GitHub repository here	None	Published in the PreProd GitHub repository (tbd)	Published in the Sandbox GitHub repository here
Training	Job aids are published in the Digital Services Implementation Portal under the Training /Sandbox folder here **Note**: Publishing of new/updated job aids may lag deployment of new code to the Preview environment	Demo material is provided to participants on an as needed basis	Draft training material is provided to participants	Job aids are published in the Digital Services Implementation Portal under the Training /Sandbox folder here **Note**: Publishing of new/updated job aids may lag deployment of new code to the Sandbox environment

Table 1 – User Feedback Environments

5.12.1 Load and Performance Tests (Performance)

Load and performance tests will be performed at regular intervals and at each release. Load testing evaluates the robustness of the application under higher-than-expected loads.

- Test the application under realistic load (traffic) to establish a baseline
- Test the load in excess of expected traffic levels by a particular percentage taking into account daily/monthly peaks

Performance testing identifies bottlenecks in a system.

- Test the stability and responsiveness of the application

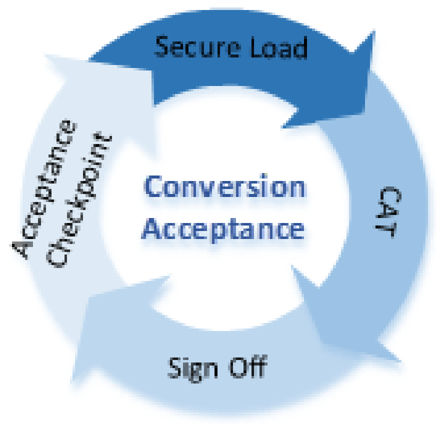
Load and performance tests must run on environments that mimic production.

- Create load and performance tests that can be run with the CI builds as you add more functionality

The Statement of Work (SOW) for all development teams includes the following requirements for Load and Performance Testing:

- The Contractor shall create and execute load and performance tests at regular intervals, and at each release.
- The Contractor shall provide a summary of all load and performance test results.

5.13 Data Conversion Acceptance Tests (DCAT)



Data conversion acceptance is the process data owners follow to validate the completeness and accuracy of the data conversion prior to its acceptance and load into the Production environment. Before a conversion can “move” into acceptance, it should meet a set of standards checkpoints indicating that it is ready for final validation. Once these checkpoints are met, the converted data is loaded into a secure user environment where data owners perform Data Conversion Acceptance Testing (DCAT) by running a series of end to end functional tests to validate the completeness and accuracy of the converted data in the day to day world. Defects are tracked and remediated until the data owner agrees that the data is acceptable to load into the Production environment. Acceptance tasks that can be converted to user stories include:

Conversion Acceptance Activities	Obtain approval for acceptance criteria checkpoints
	Request load of converted data into the user test environment
	Data owners perform conversion acceptance testing
	Provide conversion defect resolution & coordinate retests
	Obtain sign off of converted data

Conversion acceptance requires coordination between the Data Conversion Team, a software testing entity and County data owners who will perform the acceptance testing. The Project approach to software testing is still under development. This plan will be updated to conform to the Project’s testing approach once it has been defined.

5.13.1 DCAT Checkpoints

Conversion acceptance is the process for final validation and acceptance of the conversion data in the CWS-NS system. A conversion moves from conversion execution to conversion acceptance once it meets the following acceptance standards:

1. Record/data reconciliation identifies the disposition for 100% of the extracted data - meaning all of the converted data from the extract files is accounted for but not necessarily converted.
2. Exceptions reports list only low or cosmetic severity issues.
3. Critical, high and medium defects found during conversion validation testing are closed and the last round of automated testing did not result in new defects.

The Data Conversion Team will ask data owners to accept user stories associated with each of the conversion checkpoints for each data conversion. Acceptance of these user stories will signify the entry into DCAT.

5.13.2 DCAT Secure Load

The Data Conversion Team will perform the following tasks to promote converted data into the user test environment:

1. Notify the Software Test Team to work with data owners and SMEs to create DCAT user stories. The purpose of DCAT is to verify the converted data will not create risks to the Counties as they perform their daily work. To this end, DCAT should focus on end to end functional testing of daily, weekly and monthly user activities using converted data to perform them.
2. Notify the DevOps and Release Management teams to prepare the user test environment with the most current software releases and database.
3. Notify the Digital Services and TPT teams of the DCAT start date to ensure it does not interfere with any other user testing activities.
4. Notify DevOps to load converted data into the user test environment.
5. Notify data owners, Implementations and the Software Test Team that DCAT is underway. Users may need training or support to use the new system and/or enter defects.

5.13.3 Data Owner Testing

Data owners conduct testing and record defects through the Project's Defect Management Process.

5.14 Security Tests (Development, Integration PreProd)

Security testing is primarily broken out into two different types of testing:

- **White box testing** - validates whether code works as expected and verifies implemented security functionality to uncover exploitable vulnerabilities
  - Performed based on the knowledge of how the system is implemented
  - Includes analyzing data flows, code, and exception and error handling within the system, to test the intended and unintended software behavior

  - Requires access to the source code.
  - Can be performed any time during the development life cycle, it is a good practice to perform white box testing during the unit testing phase.
- **Black box testing** - identifies potential security vulnerabilities of the solution without reference to its internal workings
  - Performed using tools that allow testers to conduct security scans for both known and unknown security vulnerabilities (e.g., improper input validation, parameter injection and overflow, SQL injection attacks)
  - Can utilize “red teaming” exercises to attempt to penetrate the solution and test its security measures

The following table describes the different types of security testing that will be conducted throughout the CD pipeline

Type	Responsible Team(s)	Environment	Tools	Approach
White Box Testing - Static security code analyses	Back-end teams (e.g., TPT1, TPT2)	Development	SonarQube with Findbugs plugin	1. SonarQube is integrated into each development team's Jenkins CI build process 2. As part of any CI build process, the Jenkins runs the SonarQube security tests 3. The security test results are made available in build's Jenkins dashboard 4. Jenkins **fails** the build if new security vulnerabilities are found
	Front-end teams (e.g., Intake, CALS)	Development	RuboCop	1. RuboCop is integrated into each development team's Jenkins CI build process 2. As part of any CI build process, the Jenkins runs the RuboCop security tests 3. The security test results are made available in build's Jenkins dashboard. 4. Jenkins **fails** the build if new security vulnerabilities are found
Black-box Testing - Dynamic security code analyses	Front-end teams (e.g., Intake, CALS) and DevOps	Integration	OWASP Zed Attack Proxy (ZAP)	1. The ZAP plugin https://wiki.jenkins-ci.org/display/JENKINS/zap+plugin is added to Jenkins. 2. As part of any deployment to Integration DevOps will run the Jenkins job that starts ZAP and runs acceptance test for the application through ZAP (http-proxy). 3. ZAP collects URLs and does both passive and active scanning. 4. The ZAP reports are made available in DevOps Jenkins dashboard. 5. Jenkins **fails** the build based on level of issues found by ZAP
Black Box Testing - Infrastructure	Information Security Lead	(Live -1) this is currently Performance	Debian 8 Linux system configured with the Kali Offense Tool Set: https://tools.kali.org/tools-listing	1. Infrastructure tests are done following the Offense Security Methodology: https://www.offensive-security.com/information-security-certifications/oscp-offensive-security-certified-professional/ 2. Any identified security issues are entered into the appropriate digital

				service team's backlog as a bug story.
Black Box Testing - Application	Information Security Lead	(Live -1) this is currently Performance	Debian 8 Linux system configured with the Kali Offense Tool Set: https://tools.kali.org/tools-listing	1. Application tests are done following the OWASP testing guide: https://www.owasp.org/images/1/19/OTGv4.pdf 2. Any identified security issues are entered into the appropriate digital service team's backlog as a bug story.