Semantic Versioning and Automated Build Assembly

Brian Davenport

John Mallinger

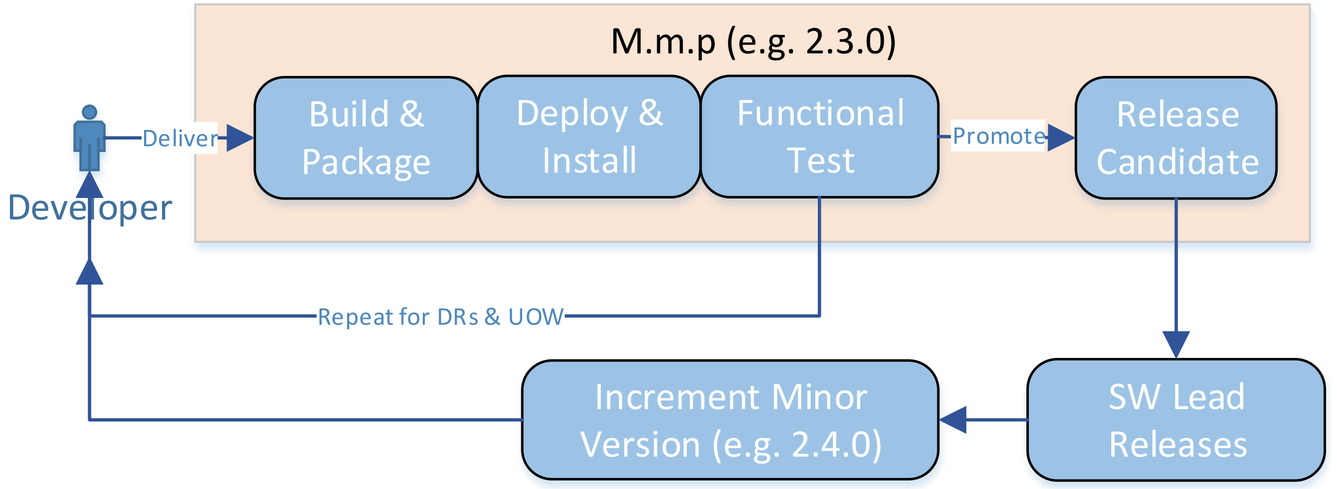
Duane Spence

Build Assembly Introduction

- DevOps Paradigm: Build once and deploy many times – don’t rebuild software that hasn’t changed
- Semantic Versioning – identification strategy that indicates type of changes and build compatibility included in a build by using defined rules and naming convention
- System Build – assembled set of compatible component and configuration item builds along with products necessary to support install, deploy, and configuration.

Semantic Versioning: An Overview

- Build Once – do not rebuild if nothing has changed
- Version identifier drivers indicator of types of changes in a build
  - Major version: change breaks compatibility with external consumers
  - Minor version: new capabilities delivered, but change is compatible to external consumers
  - Patch version: updates and discrepancy fixes in existing capabilities



1) If Developer introduces an incompatible change, they update Major version # & reset the Minor & Patch version #s (M+1).0.0 (e.g. 3.0.0)

2) If a patch is required to previously released version, the developer increments Patch version # M.m.(p+1) (e.g. 2.3.1)

3) Version is same across all products built at the same time (e.g. EIS sessions all have same version)

Drivers for System Assembly

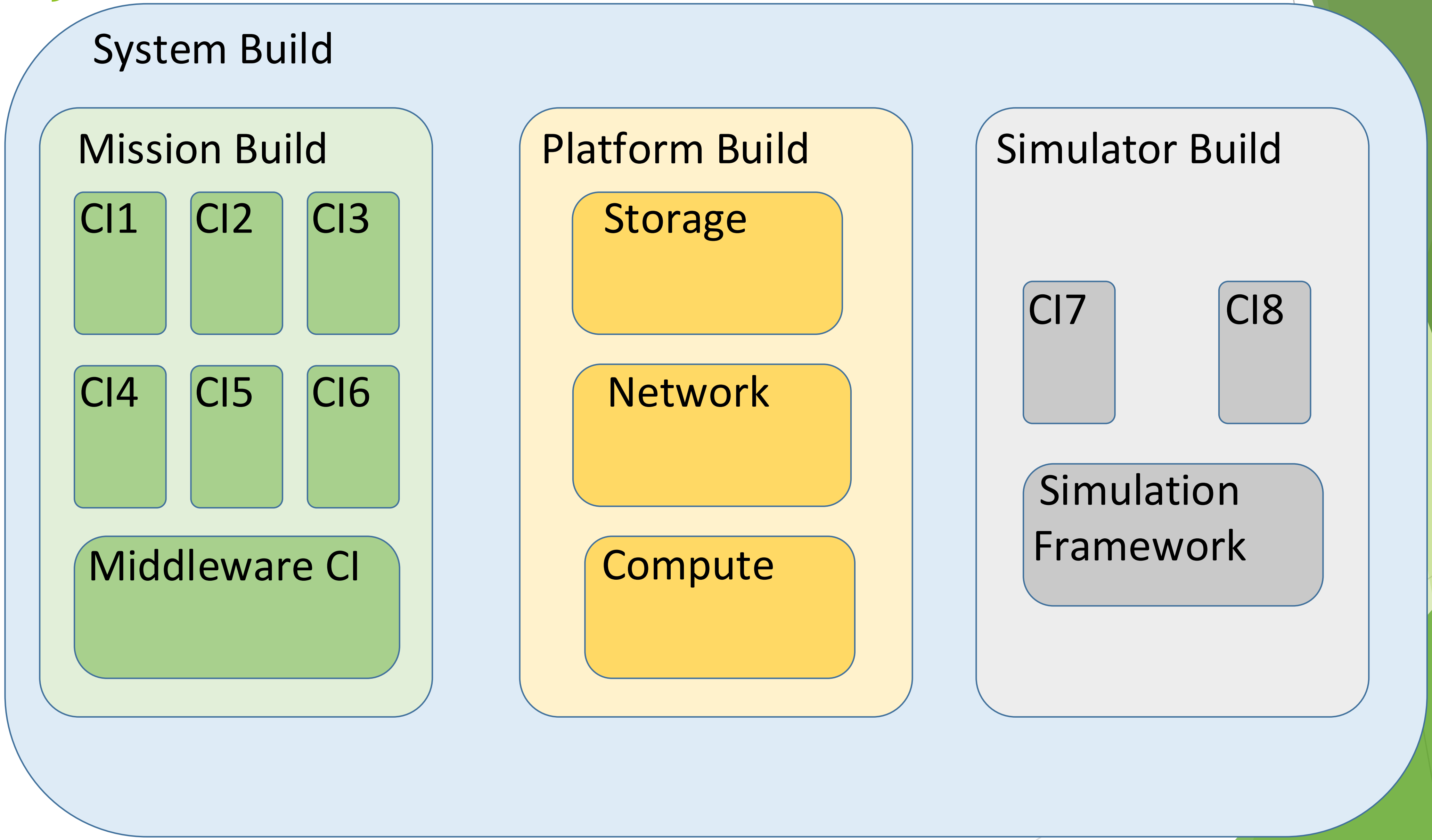
- The problem: Drowning in DevOps builds
  - 8 Configuration Items
  - 160 Custom Software Components
  - 800+ builds per week
  - 200+ COTS/FOSS products
- After we build and test individual components, how do we assemble into a system?
- A component changed – what do we need to build and retest?

System Assembly Pipeline

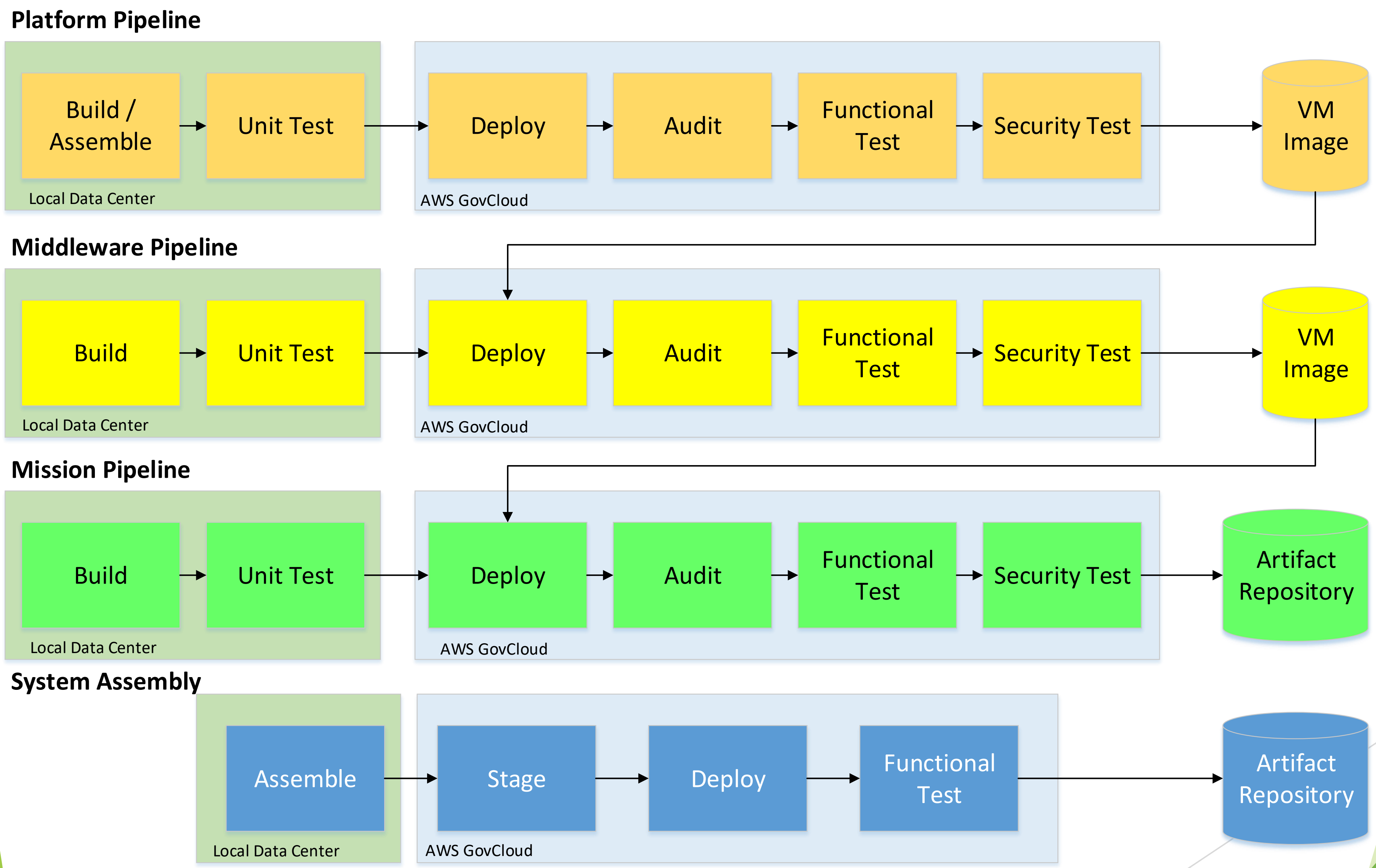
- Automated selection of component builds
  - Incorporates outputs of component pipelines
  - Test results tracking from component pipelines
  - Compatibility Validation
  - Tailoring build outputs for target deploy environments

- Automated Deploy & Checkout
  - Stage and deploy system build into representative environment
  - Run integration thread tests to validate compatibility and core functionality
  - Tag system build with test results and assess viability for promotion to ops

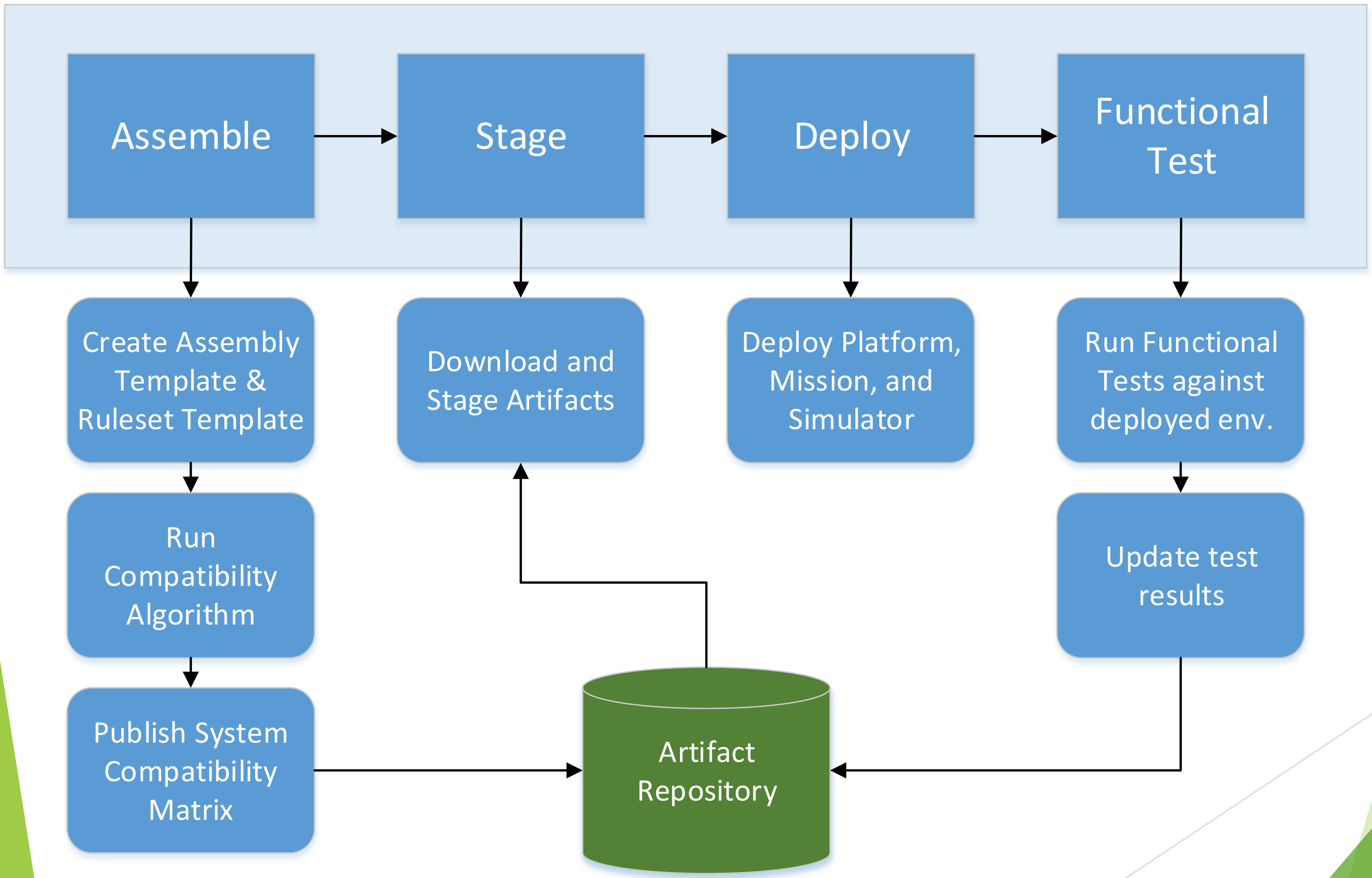
System View


System Build
Mission Build
CI1
CI2
CI3
CI4
CI5
CI6
Middleware CI
Platform Build
Storage
Network
Compute
Simulator Build
CI7
CI8
Simulation Framework

Layered Pipelines Driving CI Checkout

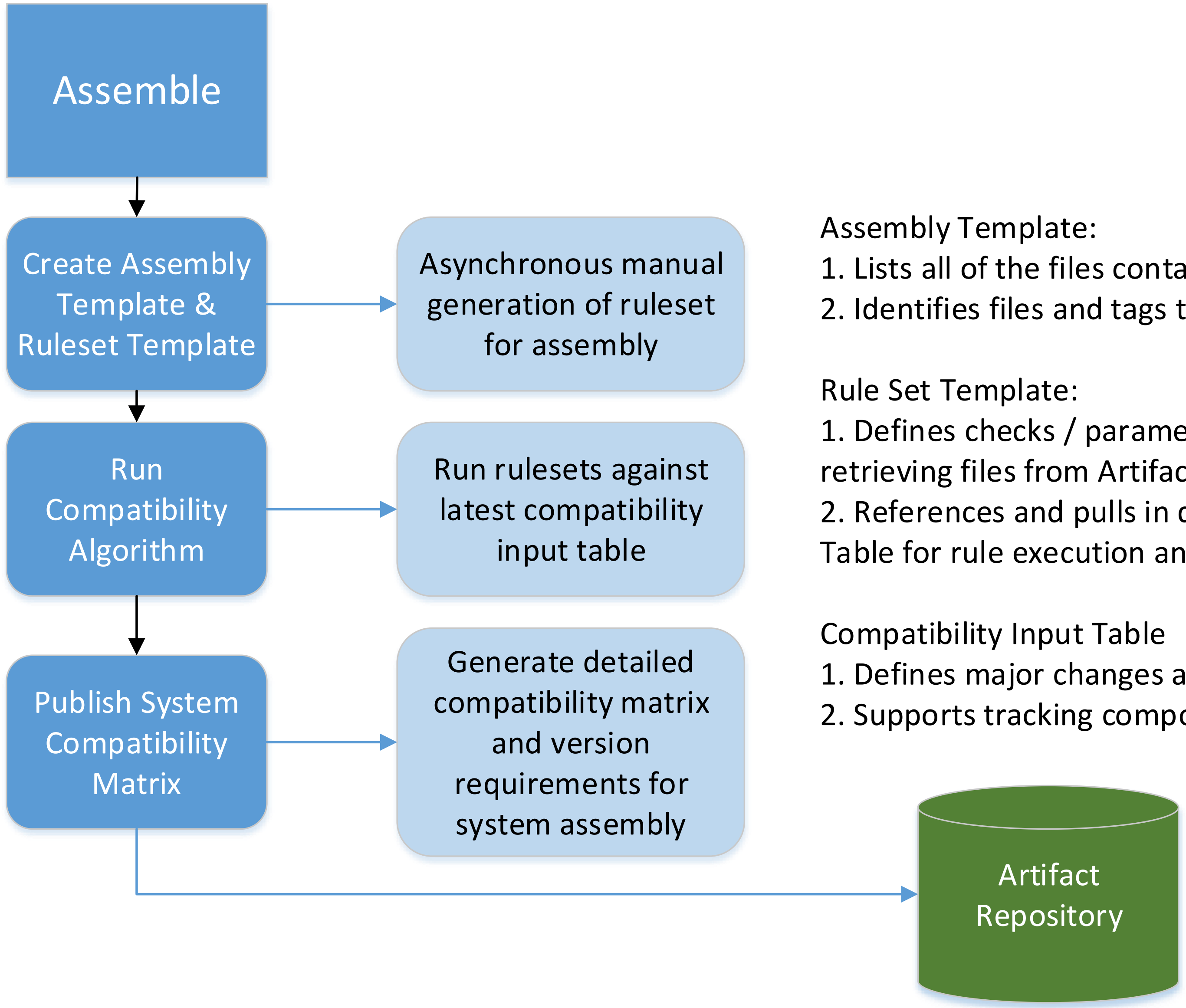


System Assembly Pipeline Overview



Assemble
Stage
Deploy
Functional Test
Create Assembly Template & Ruleset Template
Download and Stage Artifacts
Deploy Platform, Mission, and Simulator
Run Functional Tests against deployed env.
Run Compatibility Algorithm
Update test results
Publish System Compatibility Matrix
Artifact Repository

System Release: Assembly



Assemble
Create Assembly Template & Ruleset Template
Asynchronous manual generation of ruleset for assembly
Run Compatibility Algorithm
Run rulesets against latest compatibility input table
Publish System Compatibility Matrix
Generate detailed compatibility matrix and version requirements for system assembly
Artifact Repository

Assembly Template:
1. Lists all of the files contained in a release
2. Identifies files and tags to component, CI, and build

Rule Set Template:
1. Defines checks / parameters to validate when retrieving files from Artifactory repository
2. References and pulls in data from Compatibility Input Table for rule execution and enforcement

Compatibility Input Table
1. Defines major changes and impacted components
2. Supports tracking component change alignment

Case Study: Compatibility Matrix

CI1 interface change impacts CIs 2, 3, and 4

Release Compatibility Matrix
List only those Changes that impact major # of others

Change#	Status	CI1	CI2	CI3	CI4
Change1	In-Progress	3	2	1	2
Change2	In-Progress	4		2	3
Change3	In-Progress	5	3		4
Change4	In-Progress	6	4	3	5
Change5	In-Progress			4	6

Artifactory Component Versions

Date	CI1	CI2	CI3	CI4
4/16	2.5.0	1.2.0	1.0.0	1.5.0
4/18	3.0.0	2.3.0	1.0.1	1.6.0
4/19	4.0.0	2.4.0	1.1.0	2.0.0
4/20	4.0.1	2.4.1	2.0.0	3.0.0
4/25	4.0.2	2.4.2	2.1.0	3.1.0
4/30	4.1.0	2.4.3	2.2.0	4.0.0
5/1	5.0.0	2.4.4	2.3.1	4.1.0

Updates delivered and complete pipeline at different times

Release Manifest File VersionContent

Job's Run Date >>	4/18	4/19	4/22	4/30
Rel# Mission_1.0	1.0.0.23	1.1.0.34	2.0.0.43	2.1.0.53
CI1	2.5.0	3.0.0	4.0.1	4.1.0
CI2	1.2.0	2.4.0	2.4.1	
CI3	1.0.1	1.1.0	2.0.0	2.2.0
CI4	1.6.0	2.0.0	3.0.0	3.1.0

Change injected when all CIs deliver & pass pipeline checkout

Designing for System Assembly

- Identify lowest level at which a system can / should be patched
  - Structure versioning strategy and build strategy to be independent at that level
  - Needs to account for patching / update CONOPS and available outages
- Define dependencies associated with each component and CI
  - Need to capture both interfaces and APIs / shared libraries
  - Also track runtime and
- Decompose build assembly architecture
  - Strategy for rolling up and combining system builds
  - Approach for validating compatibility across components and subsystems
- Assess core set of tests needed to promote build
  - What component / CI checkouts are required as part of each lower level build?
  - What system-level tests should be completed to validate system build integration?
  - What is an acceptable risk / discovery posture for the next downstream user?

Lessons Learned

- Drive culture that supports continuous change delivery
  - Continuous deliveries of change
  - Always maintain a working build and working system
  - Requires more product oriented mindset
- Legacy CM strategies limited build and versioning at component level
  - Goal: Component level builds and patches for every mission software component
  - Issue: Extensive cross-component dependencies within a CI drove some cases of larger builds
- Design automated tests for reuse across all I&T environments
  - Automated tests can spread out beyond their initial purpose
  - Leverage common test architecture across all environments to maximize reuse and avoid refactoring