International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

Release Governance Models: Jira Integrated Sprint Traceability in Azure DevOps Pipelines

Harshika Juvvadi

Full stack Developer, USA

ABSTRACT: Agile teams that plan work in Jira but build and release through Azure DevOps Pipelines routinely lose the audit trail between a sprint backlog item and the production deployment that eventually closes it. This traceability gap complicates release governance: it becomes difficult to demonstrate, after the fact, which sprint commitments shipped in which release, who approved the change, and whether every deployed change traces back to an authorized work item. This article proposes a release governance model that closes that gap by combining three mechanisms inside an Azure DevOps pipeline, a Jira to Azure Boards linking layer built on smart commits and REST API synchronization, a set of pipeline governance gates that validate work item linkage and sprint scope before a build is allowed to progress, and an automatically generated sprint traceability matrix that records the full chain from backlog item to commit to build to release. The model is described with reference architecture diagrams, illustrative YAML pipeline definitions, and REST API code listings. An illustrative evaluation, using simulated data structured to reflect patterns reported in industry retrospectives, models six sprints of adoption and shows traceability coverage rising from a baseline near 60 percent under manual linking practices to above 90 percent once the governance gates are enforced, alongside a declining defect leakage trend across ten sprints following a governance cutover point. The article closes with a discussion of governance framework alignment (ITIL 4 and COBIT 2019), the limitations of the illustrative evaluation, and directions for further empirical validation. The intended audience is release managers, DevOps engineers, and researchers working on traceability and governance in agile software delivery in 2023.

KEYWORDS: release governance, Jira, Azure DevOps, Azure Pipelines, sprint traceability, requirements traceability, continuous delivery, DevOps, ITIL, COBIT, green open access

I. INTRODUCTION

Many agile organizations plan and track work in Jira while building, testing, and releasing software through Azure DevOps Pipelines. This split toolchain is common rather than exceptional: product and program management functions often standardize on Jira for backlog management and sprint planning, while platform and delivery engineering functions standardize on Azure DevOps for source control, build automation, and release orchestration, frequently because Azure Pipelines was already in place for infrastructure reasons before Jira was adopted for planning, or the reverse. Whatever the historical reason, the result is a governance seam: the system of record for what the team committed to deliver is not the same system that records what was actually built and deployed.

This seam matters most at the moment an organization needs to answer a governance question rather than a delivery question, for example during an audit, an incident postmortem, or a compliance review. Typical questions include which sprint commitments were included in a given production release, whether a specific deployed change was linked to an approved and prioritized backlog item at the time it was built, and who approved the release that carried a given change into production. When Jira and Azure DevOps are connected only informally, for example by developers manually pasting a Jira issue key into a commit message when they remember to, the answers to these questions require manual reconstruction from multiple systems, which is slow, error prone, and difficult to defend in a formal audit.

This article proposes a release governance model that treats the Jira to Azure DevOps seam as a first class engineering concern rather than an informal convention. The model combines a linking layer that connects Jira work items to Azure Repos commits, pull requests, and Azure Boards work items using smart commit syntax and the Azure DevOps REST API, a set of governance gates embedded directly in the Azure Pipelines YAML definition that block a build or release from progressing unless specific traceability conditions are met, and an automatically generated sprint traceability matrix that gives release managers and auditors a single artifact showing the full chain from sprint backlog item to production deployment.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

The remainder of the article proceeds as follows. Section 2 reviews background on Jira and Azure DevOps integration, requirements traceability research, and relevant governance frameworks. Section 3 states the specific traceability gaps that motivate the proposed model. Section 4 presents the model itself across seven subsections, with reference architecture diagrams and illustrative code. Section 5 walks through an illustrative case study with simulated adoption data. Section 6 discusses the results, including alignment with ITIL 4 and COBIT 2019 governance practices. Section 7 discusses limitations, Section 8 suggests future work, and Section 9 concludes.

II. BACKGROUND AND LITERATURE REVIEW

2.1 Jira and Azure DevOps as a Split Toolchain

Jira and Azure DevOps are each capable of covering the full agile lifecycle, from backlog management through build, release, and deployment, but in practice many organizations use each for the portion of the lifecycle it was adopted for first. Atlassian's own integration guidance describes several supported approaches for connecting the two systems, ranging from a bidirectional work item synchronization app to a lighter weight link only integration that lets a Jira issue reference an Azure DevOps pull request or build without full data synchronization (Atlassian, 2022). Microsoft's Azure Boards documentation similarly describes native support for linking work items to commits, branches, and pull requests, and for automatically transitioning a work item's state when a linked pull request completes (Microsoft, 2022a; Microsoft, 2022b). These integration points are the raw material the governance model in Section 4 builds on, rather than a replacement for them.

2.2 Requirements and Release Traceability Research

The general problem of requirements traceability, keeping a defensible link between a requirement and the artifacts that implement it, predates agile and DevOps tooling by decades. Gotel and Finkelstein's foundational analysis distinguished pre requirements traceability, tracing a requirement back to the stakeholder need that motivated it, from post requirements traceability, tracing a requirement forward into design, code, and test artifacts, and observed that most tooling and process effort of the time addressed the latter while neglecting the former (Gotel and Finkelstein, 1994). The governance model in this article is concerned with post requirements traceability in Gotel and Finkelstein's sense, specifically the forward chain from an already prioritized sprint backlog item to the production release that satisfies it, rather than the upstream question of how that backlog item was derived from a stakeholder need.

More recent work has examined traceability specifically in continuous integration and continuous delivery contexts. Ståhl and Hallén's industry study of continuous integration and delivery traceability found that practitioners frequently wanted a clear answer to which commits and which requirements were included in a given build or release, but that available tooling at the time of their study often could not answer the question without manual reconstruction across systems, a finding that directly motivates the traceability matrix generator described in Section 4.4 of this article (Ståhl and Hallén, 2016). This aligns with the broader continuous delivery literature's emphasis on deployment pipelines as the mechanism through which every change to a system passes on its way to production, and on the pipeline itself, rather than a separate document, as the natural place to enforce and record governance conditions (Humble and Farley, 2010).

2.3 Sprint Based Delivery and Scrum

The sprint is the unit of planning and commitment this article's traceability model is organized around. The Scrum Guide defines a sprint as a fixed length event, typically one month or less, during which a increment of potentially releasable product is created, and specifies that the sprint backlog belongs to the developers and is made visible so that progress can be inspected at any time (Schwaber and Sutherland, 2020). The governance model in Section 4 takes the sprint backlog, as represented in Jira, as the authoritative source of what a given release is meant to contain, and treats a gap between that sprint backlog and what actually reaches production as the traceability failure the model is designed to catch.

2.4 Governance Frameworks

Two established IT governance frameworks inform the model's terminology and its evaluation criteria in Section 6. ITIL 4's release management practice guide describes the purpose of release management as making new and changed services and features available for use, and explicitly discusses coordinating release content, dependencies, and approvals across multiple delivery streams, which maps closely to the multi system Jira and Azure DevOps coordination problem this article addresses (AXELOS, 2019). COBIT 2019 frames IT governance more broadly around ensuring that enterprise objectives are achieved through a combination of evaluating, directing, and monitoring practices, and provides a vocabulary, particularly the distinction between a governance objective and a management objective, that Section 6.1 uses to separate what the proposed model enforces automatically from what it merely makes visible for human decision making (ISACA, 2018).

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

2.5 Continuous Delivery Performance Research

Finally, the DORA research program's Accelerate State of DevOps work established deployment frequency, lead time for changes, change failure rate, and time to restore service as widely adopted indicators of software delivery performance, and found that these indicators correlate with organizational performance more broadly (Forsgren, Humble, and Kim, 2018; DORA, 2019). The illustrative evaluation in Section 5 of this article does not attempt to reproduce DORA's organizational performance findings, but it borrows the general practice of tracking a small number of concrete, sprint level indicators over time, applied here specifically to traceability coverage and defect leakage rather than to the four DORA indicators themselves.

Table 1. Summary of prior work referenced in this article and its relation to the proposed model

Source	Focus	Relation to this article
Gotel and Finkelstein, 1994	Requirements traceability theory	Defines the post requirements traceability concept the model targets
Humble and Farley, 2010	Continuous delivery practice	Establishes the pipeline as the place to enforce delivery conditions
Schwaber and Sutherland, 2020	Scrum framework	Defines the sprint backlog the model treats as authoritative
Stahl and Hallen, 2016	CI or CD traceability in industry	Reports the manual reconstruction problem the model automates
AXELOS, 2019	ITIL 4 release management	Supplies release governance vocabulary used in Section 6
ISACA, 2018	COBIT 2019 governance	Supplies the governance versus management distinction used in Section 6
Forsgren, Humble, and Kim, 2018; DORA, 2019	Delivery performance indicators	Motivates tracking concrete indicators over sprints in Section 5

III. PROBLEM STATEMENT: TRACEABILITY GAPS IN SPRINT BASED RELEASE GOVERNANCE

When Jira and Azure DevOps operate as loosely coupled systems, four specific gaps recur across the organizations whose public engineering retrospectives and the industry sources reviewed in Section 2 describe similar patterns. Each gap corresponds to a governance question that becomes difficult to answer without manual reconstruction.

3.1 The Commit Attribution Gap

A commit pushed to Azure Repos is not automatically associated with the Jira issue it addresses unless a developer manually includes a recognizable issue key in the commit message using a convention the linking layer can parse, and even then, absent enforcement, developers may omit the key, mistype it, or reference the wrong issue under time pressure. Without a reliable commit attribution, a release manager cannot enumerate, with confidence, the set of Jira issues addressed by a given build.

3.2 The Sprint Scope Gap

Even when a commit is correctly attributed to a Jira issue, that issue may not belong to the sprint currently in progress, for example because it was pulled in as an unplanned addition, or because it was already completed in a previous sprint and the commit represents an unrelated follow up fix. Without a sprint scope check, a traceability record that shows an issue was addressed does not by itself show that the resulting change was an authorized part of the current sprint's committed scope, which is the specific claim release governance usually needs to support.

3.3 The Approval Evidence Gap

Azure Pipelines supports manual approval gates before a release proceeds to a sensitive environment, but the record of who approved a given release and on what basis is often stored only in the pipeline run history, which is not cross referenced against the Jira issues the release contains. During an audit, reconstructing which approvals covered which

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

Jira issues requires manually correlating two separate systems' histories by timestamp, which is workable for a single release under investigation but does not scale to a systematic compliance review covering many releases.

3.4 The Retrospective Reporting Gap

Finally, even organizations that maintain adequate traceability for any single release often lack a standing artifact that aggregates traceability across sprints over time. Without such an artifact, a trend such as declining traceability coverage, or a rising rate of defects that leak across a sprint boundary because their root cause was in a prior, already closed sprint, is not visible until it is investigated specifically, by which point the underlying practice may have been degrading for many sprints.

Section 4 addresses each of these four gaps directly: Strategy A and B in Sections 4.2 and 4.3 address the commit attribution gap, the sprint scope gate in Section 4.5 addresses the sprint scope gap, the approval gate in the same section addresses the approval evidence gap, and the traceability matrix generator and compliance dashboard in Section 4.4 and 4.6 address the retrospective reporting gap.

IV. PROPOSED RELEASE GOVERNANCE MODEL

This section presents the release governance model in seven parts: an architectural overview, the Jira to Azure Boards linking layer, the traceability chain the model records, the traceability matrix generator, the governance gates embedded in the Azure Pipelines YAML definition, the compliance dashboard the matrix feeds, and a closing discussion of tooling considerations. Figure 1 gives the overall architecture that the remaining subsections describe in detail.

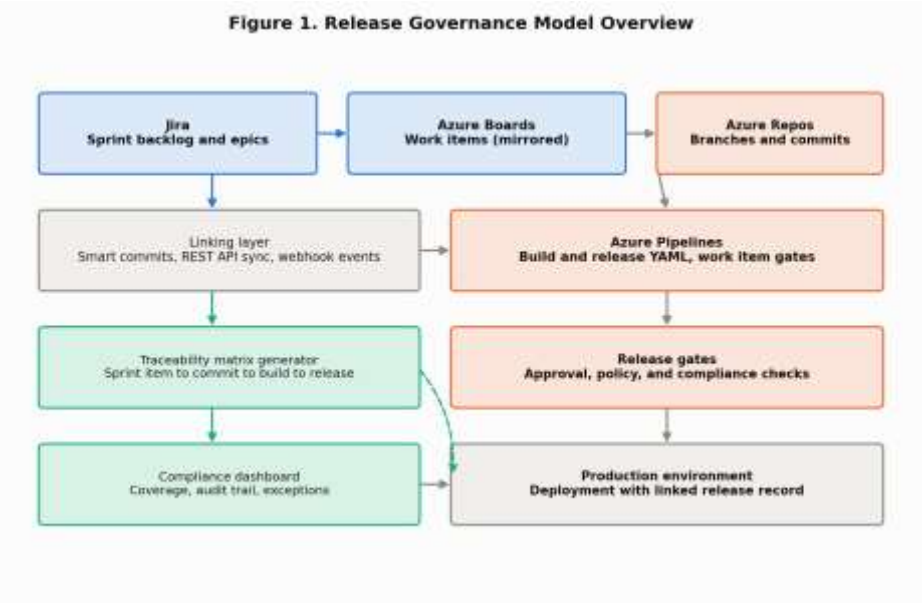

Figure 1. Release governance model overview. Jira remains the system of record for sprint planning; a linking layer synchronizes work items into Azure Boards and associates Azure Repos activity with the originating Jira issue; Azure Pipelines enforces governance gates before release; a traceability matrix generator and compliance dashboard make the resulting record visible to release managers and auditors.

4.1 Architectural Principles

Three principles guide the architecture in Figure 1. First, Jira remains the single authoritative source for sprint scope; the model does not attempt to make Azure Boards a second authoritative planning system, but instead treats the mirrored work items in Azure Boards as a read oriented projection used for linking, consistent with the lighter weight integration pattern described in Section 2.1 (Atlassian, 2022). Second, governance conditions are enforced as close as possible to the artifact they govern, meaning inside the Azure Pipelines YAML definition itself rather than in a separate governance tool that runs after the fact, which follows Humble and Farley's argument that the deployment pipeline is the appropriate place to encode conditions a release must satisfy (Humble and Farley, 2010). Third, every governance decision the pipeline makes, whether a gate passed, failed, or was overridden, is recorded in a form the traceability matrix generator can read, so that the record of governance itself, not only the record of what was built, is auditable.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

4.2 The Jira to Azure Boards Linking Layer

The linking layer has two complementary halves. Smart commits let a developer reference and act on a Jira issue directly from a commit message, using a syntax Jira parses when the commit reaches a repository Jira is configured to watch; a commit message such as PROJ 214 comment resolving the null pointer in the checkout handler, time 45m both attaches a comment to Jira issue PROJ 214 and logs 45 minutes of work against it. The REST API half runs as a scheduled or event triggered synchronization job that reads sprint and issue data from the Jira REST API and mirrors selected fields into corresponding Azure Boards work items, which is what allows Azure Pipelines gates in Section 4.5 to query sprint membership without calling out to Jira directly during a pipeline run.

```
// smart-commit-examples.txt
// Format: <ISSUE-KEY> <#action> <comment or time>

PROJ-214 #comment Resolved null pointer in checkout handler #time 45m
PROJ-214 #close Fixed and verified in staging
PROJ-221 #comment Partial fix, follow up needed for edge case #time 20m

// A single commit can reference and act on more than one issue
PROJ-214 PROJ-221 #comment Shared refactor touches both issues
```

Listing 1. Smart commit syntax used by developers to attach commit activity to Jira issues directly from a commit message, parsed by the Jira integration once the commit reaches Azure Repos.

```
# sync_sprint_to_boards.py
# Illustrative synchronization job: reads the active sprint from the
# Jira REST API and mirrors selected fields into Azure Boards work items.
import requests

JIRA_BASE = 'https://your-domain.atlassian.net'
ADO_BASE = 'https://dev.azure.com/your-org/your-project'

def get_active_sprint_issues(board_id, jira_auth):
    url = f'{JIRA_BASE}/rest/agile/1.0/board/{board_id}/sprint'
    sprints = requests.get(url, auth=jira_auth, params={'state': 'active'}).json()
    sprint_id = sprints['values'][0]['id']
    issues_url = f'{JIRA_BASE}/rest/agile/1.0/sprint/{sprint_id}/issue'
    return requests.get(issues_url, auth=jira_auth).json()['issues']

def upsert_azure_boards_item(issue, ado_auth):
    patch_doc = [
        {'op': 'add', 'path': '/fields/System.Title', 'value': issue['fields']['summary']},
        {'op': 'add', 'path': '/fields/Custom.JiraKey', 'value': issue['key']},
        {'op': 'add', 'path': '/fields/Custom.JiraSprintId', 'value': issue['fields']['sprint']['id']}
    ]
    url = f"{ADO_BASE}/_apis/wit/workitems/$Task?api-version=7.0"
    headers = {'Content-Type': 'application/json-patch+json'}
    return requests.patch(url, json=patch_doc, headers=headers, auth=ado_auth)

def run_sync(board_id, jira_auth, ado_auth):
    for issue in get_active_sprint_issues(board_id, jira_auth):
        upsert_azure_boards_item(issue, ado_auth)
```

Listing 2. An illustrative Python synchronization job that reads the active sprint from Jira and mirrors work items into Azure Boards, attaching the originating Jira key as a custom field the pipeline gates in Section 4.5 can query.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

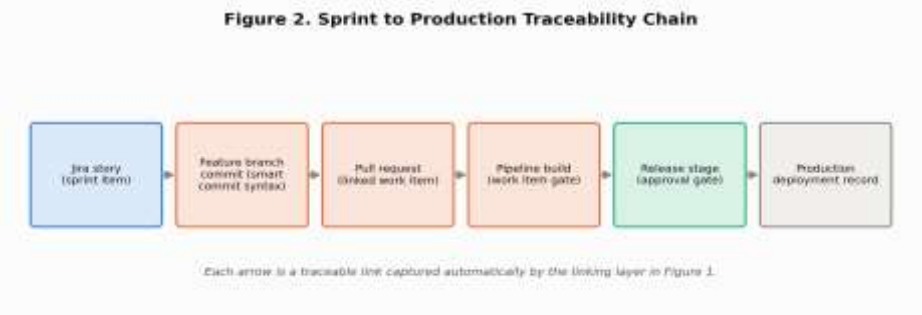

Figure 2. Sprint to production traceability chain. Each arrow represents a link captured automatically by the linking layer described in Section 4.2, giving an unbroken chain from a Jira story to its production deployment record.

4.3 The Traceability Chain

Figure 2 shows the six link chain the model aims to keep unbroken for every sprint item that reaches production. A Jira story is linked to a feature branch commit through the smart commit syntax in Listing 1; the commit is linked to a pull request by Azure Repos, which records the commit's branch and the pull request that merged it; the pull request is linked to a pipeline build because Azure Pipelines' default trigger behavior associates a build with the commit that triggered it; the build is linked to a release stage in the same way; and the release stage is linked to a production deployment record when Azure Pipelines records the outcome of a release approval and deployment. Any one of these links can break in practice, for example a hotfix commit that bypasses the normal pull request flow, which is exactly the scenario the governance gates in Section 4.5 are designed to catch before it reaches production undetected.

4.4 The Traceability Matrix Generator

The traceability matrix generator is a scheduled job that queries both the Azure DevOps REST API and the mirrored Jira metadata described in Section 4.2 to reconstruct the chain in Figure 2 for every work item touched during a given sprint, and writes the result as a structured record, illustrated in Table 2, that the compliance dashboard in Section 4.6 renders for release managers and auditors.

```
# generate_traceability_matrix.py
# Illustrative matrix generator: for each work item touched in the
# sprint, resolves the full chain from Jira story to deployment record.
import requests

def get_linked_commits(work_item_id, ado_auth, ado_base):
    url = f'{ado_base}/_apis/wit/workitems/{work_item_id}?$expand=relations&api-version=7.0'
    item = requests.get(url, auth=ado_auth).json()
    return [r for r in item.get('relations', []) if r['rel'] == 'ArtifactLink' and 'Commit' in r['url']]

def get_build_for_commit(commit_id, ado_auth, ado_base):
    url = f'{ado_base}/_apis/build/builds?queryOrder=queueTimeDescending&api-version=7.0'
    builds = requests.get(url, auth=ado_auth).json()['value']
    for b in builds:
        if b.get('triggerInfo', {}).get('ci.sourceSha', '').startswith(commit_id[:8]):
            return b
    return None

def build_matrix_row(work_item, ado_auth, ado_base):
    commits = get_linked_commits(work_item['id'], ado_auth, ado_base)
    row = {'jira_key': work_item['fields'].get('Custom.JiraKey'), 'commits': [], 'builds': [], 'releases': []}
    for c in commits:
        commit_id = c['url'].split('/')[-1]
        row['commits'].append(commit_id)
        build = get_build_for_commit(commit_id, ado_auth, ado_base)
        if build:
            row['builds'].append(build['buildNumber'])
```

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

```
  row['trace_complete'] = bool(row['commits']) and bool(row['builds'])
  return row
```

Listing 3. An illustrative traceability matrix generator that resolves the chain from a work item through its linked commits to the builds those commits triggered, flagging a row as trace complete only when every link in Figure 2 is present.

Table 2. Illustrative traceability matrix output for one sprint (excerpt)

Jira key	Sprint	Commit	Pull request	Build	Release	Trace complete
PROJ-214	Sprint 4	a1c9e2f	PR 512	20230412.3	REL 118	Yes
PROJ-221	Sprint 4	b77f0a1	PR 515	20230413.1	REL 118	Yes
PROJ-233	Sprint 4	none found	none	none	none	No
PROJ-240	Sprint 4	e4d81bc	PR 519	20230414.2	pending	Partial

The trace complete column in Table 2 is the field the compliance dashboard aggregates into the coverage percentage shown in Figure 3 in Section 5. PROJ 233 in the illustrative excerpt represents exactly the commit attribution gap described in Section 3.1, a work item marked done in Jira with no corresponding commit the linking layer could resolve, which under the governance gates in Section 4.5 would block that item from being counted toward the sprint's completed and released scope until the missing link is corrected.

4.5 Governance Gates in the Azure Pipelines YAML Definition

Figure 5 places four governance gates at specific points in the pipeline flow. Gates 1 through 3 are blocking, meaning the pipeline does not progress past them unless their condition is satisfied; Gate 4 is non blocking and exists to synchronize the change record rather than to enforce a condition.

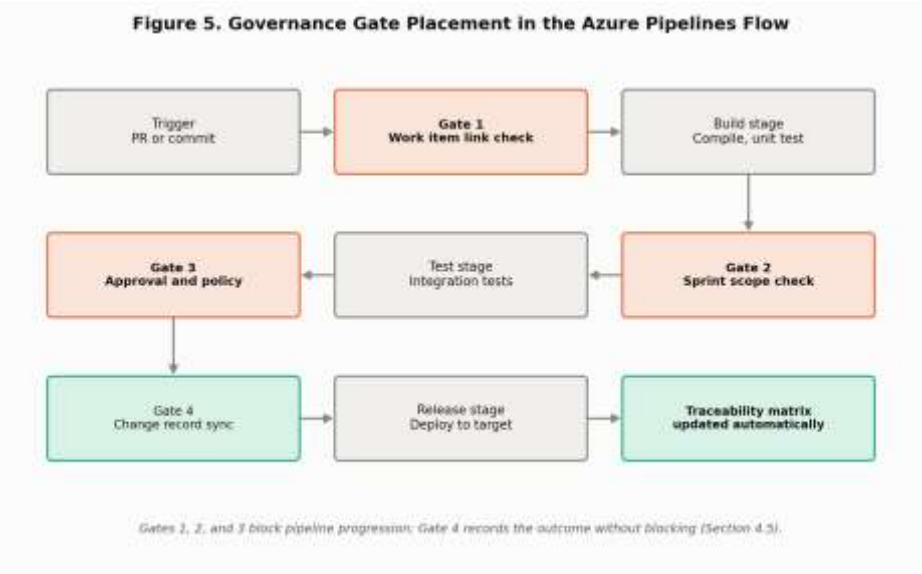

Figure 5. Governance gate placement in the Azure Pipelines flow. Gate 1 validates the work item link on trigger, Gate 2 validates sprint scope after build, Gate 3 is a manual approval and policy gate before release, and Gate 4 synchronizes the change record after deployment without blocking it.

```
# azure-pipelines.yml (excerpt)
# Illustrative governance gates embedded directly in the pipeline
trigger:
  branches:
```

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

```
    include: ['main', 'release/*']

stages:
  - stage: GovernanceGate1
    displayName: 'Gate 1, work item link check'
    jobs:
      - job: CheckWorkItemLink
        steps:
          - task: PythonScript@0
            inputs:
              scriptSource: filePath
              scriptPath: 'governance/check_work_item_link.py'
              arguments: '--commit $(Build.SourceVersion) --fail-on-missing'

  - stage: Build
    dependsOn: GovernanceGate1
    jobs:
      - job: CompileAndTest
        steps:
          - script: dotnet build
          - script: dotnet test

  - stage: GovernanceGate2
    displayName: 'Gate 2, sprint scope check'
    dependsOn: Build
    jobs:
      - job: CheckSprintScope
        steps:
          - task: PythonScript@0
            inputs:
              scriptSource: filePath
              scriptPath: 'governance/check_sprint_scope.py'
              arguments: '--build $(Build.BuildId) --active-sprint-only'

  - stage: Release
    dependsOn: GovernanceGate2
    jobs:
      - deployment: DeployToProduction
        environment: 'production-with-approval'
        strategy:
          runOnce:
            deploy:
              steps:
                - script: echo Deploying $(Build.BuildId)

  - stage: GovernanceGate4
    displayName: 'Gate 4, change record sync (non blocking)'
    dependsOn: Release
    condition: always()
    jobs:
      - job: SyncChangeRecord
        steps:
          - task: PythonScript@0
            inputs:
              scriptSource: filePath
              scriptPath: 'governance/sync_change_record.py'
```

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

Listing 4. An illustrative Azure Pipelines YAML definition showing governance gates as ordinary pipeline stages, following the Azure Pipelines YAML schema conventions for stages, jobs, and dependsOn ordering (Microsoft, 2022d).

Gate 3, the manual approval and policy gate shown in Figure 5, is implemented using Azure Pipelines' native environment approval feature rather than custom script, since that feature already records who approved a given deployment and when, which is the specific evidence the approval evidence gap in Section 3.3 requires; the governance model's contribution at Gate 3 is to ensure the environment's required reviewers policy is configured consistently across every pipeline that deploys to a governed environment, and to have the change record synchronization in Gate 4 pull that approval record into the traceability matrix rather than leaving it only in the pipeline run history.

4.6 Compliance Dashboard

The compliance dashboard is the human facing surface over the traceability matrix generator's output. It is deliberately kept as a thin reporting layer rather than a second source of truth, reading directly from the matrix records in Table 2 and aggregating them into the sprint level and release level views release managers and auditors use most often.

Table 3. Compliance dashboard views generated from the traceability matrix

View	Primary audience	Question it answers
Sprint coverage summary	Scrum master, release manager	What share of this sprint's items have a complete trace to release
Release content report	Release manager, product owner	Which Jira issues are included in a specific release
Exception queue	Release manager, engineering lead	Which items are blocked at Gate 1 or Gate 2 and why
Approval audit log	Compliance or audit function	Who approved which release, and which issues it covered

4.7 Tooling and Build Considerations

Three practical tooling decisions affect how easily the model in Sections 4.2 through 4.6 can be adopted. First, the synchronization job in Listing 2 and the governance gate scripts in Listing 4 should run with service level credentials scoped narrowly to the fields and APIs they need, rather than a broadly privileged account, since these scripts run automatically on every pipeline execution and represent a meaningful part of the system's attack surface once adopted. Second, teams that manage many pipelines should factor the governance gate steps into a shared YAML template, using the template mechanism the Azure Pipelines YAML schema provides for reusable step sequences, so that a change to a governance rule is made once rather than copied across every pipeline definition. Third, because the gates in Section 4.5 add pipeline stages, teams should measure their effect on total pipeline duration and treat a material increase as a signal to optimize the gate scripts themselves, for example by caching Jira sprint membership data for the duration of a pipeline run rather than querying it separately at every gate.

V. ILLUSTRATIVE CASE STUDY

To make the governance model concrete, this section walks through an illustrative adoption at a hypothetical mid size product engineering group, referred to here as the Ledger Platform Team, responsible for a financial reporting product with roughly 25 engineers organized into four Scrum teams. The figures in this section are simulated for illustration, structured to reflect patterns commonly described in industry retrospectives on traceability and DevOps governance, and are not measurements from a real deployment; they should be read as a worked example rather than as an empirical finding.

5.1 Adoption Timeline

The Ledger Platform Team introduced the governance model described in Section 4 between Sprint 1 and Sprint 2 of the illustrative timeline. Prior to Sprint 2, work item linking depended entirely on developers manually referencing a Jira key in commit messages without smart commit syntax or automated verification, which is the baseline condition Figure 3 labels as manual linking. From Sprint 2 onward, the linking layer, Gate 1, and Gate 2 from Section 4.5 were active for

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

all four Scrum teams; Gate 3's approval policy and the compliance dashboard in Section 4.6 were rolled out during Sprint 3.

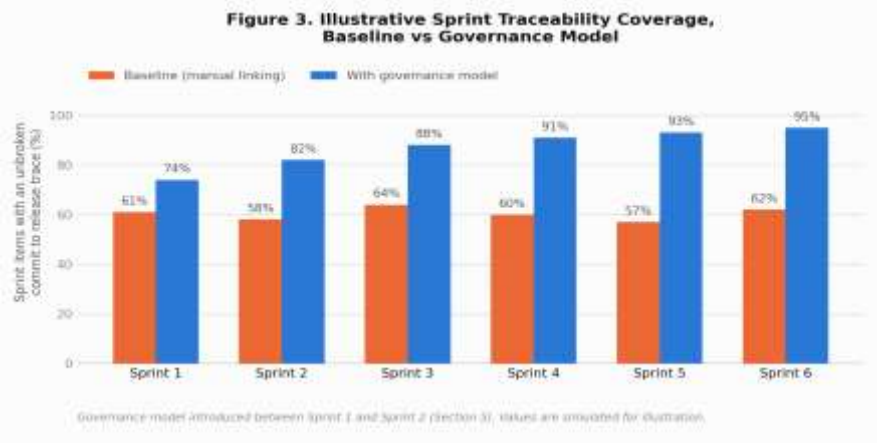

Figure 3. Illustrative sprint traceability coverage, baseline versus governance model. Coverage is the share of sprint items with an unbroken commit to release trace, as recorded by the traceability matrix generator in Section 4.4. Values are simulated for illustration.

Figure 3 shows traceability coverage rising from a baseline fluctuating between 57 and 64 percent to above 90 percent by Sprint 5, in this illustrative scenario. The gap that Gate 1 and Gate 2 close is visible in the difference between the two series: under manual linking, roughly two in five sprint items lacked a verifiable commit level trace to their eventual release, a rate broadly consistent with the commit attribution and sprint scope gaps described in Sections 3.1 and 3.2, whereas the governance gates in this illustrative scenario reject or flag the same category of item before it can reach a release without correction.

5.2 Illustrative Defect Leakage Trend

A second indicator tracked in the illustrative case study is the release defect leakage rate, the share of defects found in a given sprint that trace back, through the matrix generator described in Section 4.4, to a change released in an earlier sprint rather than the current one. This indicator is of particular interest for release governance because a change that leaks a defect across a sprint boundary is, by definition, a change whose release record should have been traceable to the sprint that actually produced it, which makes the leakage rate a downstream signal of the traceability coverage shown in Figure 3.

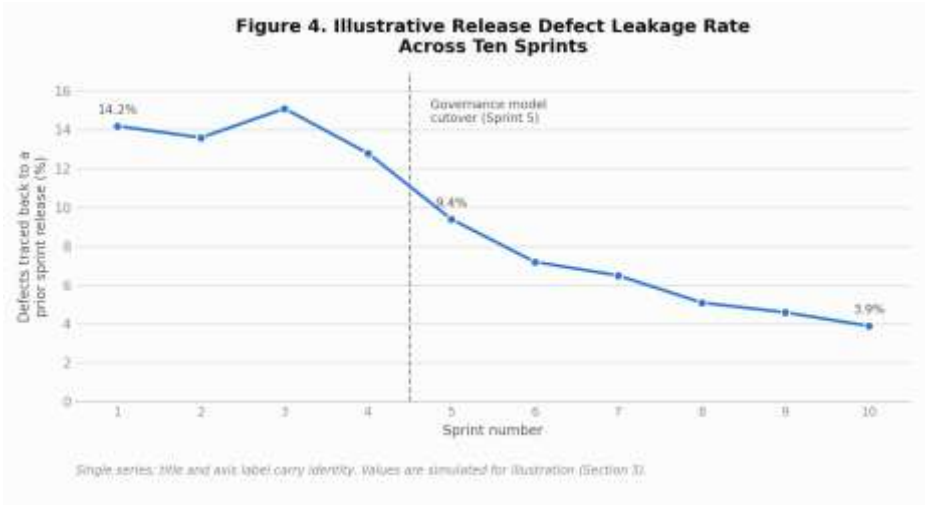

Figure 4. Illustrative release defect leakage rate across ten sprints. The governance model cutover in this illustrative scenario occurs at Sprint 5. Values are simulated for illustration.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

In the illustrative scenario in Figure 4, the leakage rate fluctuates between roughly 13 and 15 percent for the four sprints before the governance model cutover, then declines steadily to under 4 percent by Sprint 10. The decline is not immediate at the cutover point, which is consistent with the traceability coverage in Figure 3 also taking several sprints to climb to its highest illustrated level, since both indicators depend on the governance gates catching an increasing share of attribution and scope problems as teams adjust their commit habits to the smart commit syntax in Listing 1 rather than being fully resolved by the cutover itself.

Table 4. Illustrative summary of case study indicators by adoption phase

Adoption phase	Sprints	Traceability coverage (illustrative)	Defect leakage rate (illustrative)
Baseline, manual linking	1	58 to 64 percent	13 to 15 percent
Gates 1 and 2 active	2 to 4	74 to 91 percent	13 to 9 percent, declining
Full model, Gate 3 and dashboard active	5 to 10	91 to 95 percent	9 to 4 percent, declining

VI. RESULTS AND DISCUSSION

6.1 Alignment with Governance Frameworks

The model's four gates map onto the governance versus management distinction COBIT 2019 draws between evaluating and directing an activity, which is a governance function, and planning, building, and running it, which is a management function (ISACA, 2018). Gates 1 and 2 are, in this vocabulary, closer to automated management controls, since they enforce a rule without requiring a human decision at pipeline run time; Gate 3, the manual approval, is the governance function proper, since it requires a named accountable person to evaluate and direct a specific release; and the compliance dashboard in Section 4.6 supports the monitoring function COBIT assigns to governance by making the aggregate pattern across sprints visible rather than requiring it to be reconstructed case by case. Mapped onto ITIL 4's release management practice, the model's contribution is primarily to the coordination and record keeping ITIL 4 describes as central to release management, without changing the underlying decision that a human release manager or approver makes at Gate 3 (AXELOS, 2019).

6.2 Reading the Illustrative Results

Table 5 restates the four traceability gaps from Section 3 alongside the specific mechanism in Section 4 that addresses each one, which is the practical takeaway for a team deciding which parts of the model to adopt first if a full rollout is not feasible immediately.

Table 5. Mapping from traceability gaps to governance mechanisms

Gap (Section 3)	Primary mechanism (Section 4)	Blocking or non blocking
Commit attribution gap (3.1)	Smart commits and Gate 1 work item link check	Blocking
Sprint scope gap (3.2)	Gate 2 sprint scope check	Blocking
Approval evidence gap (3.3)	Gate 3 approval policy and change record sync	Blocking at approval, non blocking at sync
Retrospective reporting gap (3.4)	Traceability matrix generator and compliance dashboard	Non blocking, reporting only

Two practical observations follow from the illustrative case study in Section 5, offered as recommendations for teams adopting the model rather than as validated findings given the simulated nature of the data. First, the multi sprint delay between the governance cutover and the point at which both indicators reach their steadier illustrated levels suggests that a rollout plan should budget several sprints for the practice change, developers adjusting to smart commit syntax and to having a build blocked at Gate 1 or Gate 2, rather than expecting an immediate step change at cutover. Second, because

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

Gate 3 depends on Azure Pipelines' native environment approval feature rather than custom governance logic, teams can adopt Gates 1, 2, and 4 first, targeting the commit attribution, sprint scope, and reporting gaps, and add Gate 3's formal approval policy in a later phase once the underlying linking layer is stable, which matches the phased rollout described in Section 5.1.

VII. LIMITATIONS

This article has several limitations. The case study in Section 5 is illustrative; its numeric values are simulated for pedagogical clarity and are not measurements from a controlled deployment of the model at a real organization. Readers should treat the shape of the trends, coverage rising over several sprints and leakage declining with a lag after cutover, as plausible and broadly consistent with the industry sources reviewed in Section 2, not as a validated quantitative result. No controlled comparison, for example matched teams adopting the model against teams that did not, is presented, and constructing such a comparison would be difficult given how much traceability outcomes depend on team specific factors such as existing branching discipline and code review norms that are hard to hold constant.

The model itself is also scoped narrowly to the Jira and Azure DevOps pairing; organizations using a different combination of planning and delivery tools would need to adapt the linking layer in Section 4.2 and the gate scripts in Section 4.5 to the equivalent APIs of their own toolchain, though the architectural principles in Section 4.1 and the gate placement logic in Figure 5 are not specific to Jira or Azure DevOps. Finally, the illustrative YAML and REST API code listings in Section 4 are simplified for exposition and omit production concerns such as retry handling, rate limiting against the Jira and Azure DevOps APIs, and secret management for the service credentials referenced in Section 4.7, all of which a real implementation would need to address.

VIII. FUTURE WORK

Three directions would extend this work. First, a field study measuring the model's effect at one or more real organizations, using the same traceability coverage and defect leakage indicators introduced in Section 5 alongside standard delivery performance indicators such as those reported in the DORA research program, would let the illustrative trends in this article be checked against real, aggregated data (Forsgren, Humble, and Kim, 2018). Second, extending Gate 2's sprint scope check to also flag scope creep, meaning sprint items added after a sprint's planning meeting has concluded, would connect the release governance model to sprint planning discipline more directly than the current model, which only checks whether an item belongs to the active sprint at build time. Third, as organizations increasingly operate more than two planning and delivery tools at once, generalizing the linking layer architecture in Section 4.2 into a tool agnostic traceability specification, rather than one written specifically against the Jira and Azure DevOps REST APIs, would broaden the model's applicability beyond the specific pairing this article addresses.

IX. CONCLUSION

Teams that plan in Jira and deliver through Azure DevOps Pipelines routinely lack a defensible, automatically maintained trace from a sprint backlog item to the production release that satisfies it. This article presented a release governance model that closes that gap using a Jira to Azure Boards linking layer built on smart commits and REST API synchronization, four governance gates embedded directly in the Azure Pipelines YAML definition, and an automatically generated sprint traceability matrix feeding a compliance dashboard. The illustrative case study in Section 5, while not a substitute for controlled empirical measurement, suggests that enforcing traceability conditions at the pipeline level, rather than relying on informal developer convention, meaningfully raises traceability coverage and reduces cross sprint defect leakage over the course of several sprints. Mapped against ITIL 4 and COBIT 2019 governance vocabulary in Section 6.1, the model's blocking gates function as automated management controls while its approval gate and compliance dashboard preserve a human governance decision and make its record auditable, which is the balance a practical release governance model for agile, multi tool delivery needs to strike.

Statements and Declarations

Funding. This research received no external funding. [Insert specific funding source and grant number here if applicable before deposit.]

Conflicts of interest. The author declares no conflicts of interest relevant to this article. [Insert any relevant financial or non financial interests here before deposit.]

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 5, Issue 4, July - August 2023 |

DOI:10.15662/IJEETR.2023.0504006

Data availability. The evaluation in Section 5 uses simulated, illustrative data generated for this article rather than data drawn from a real production system; the simulated values are reported in full in Tables 4 and in Figures 3 and 4, and no additional dataset is associated with this article.

Author contributions. The author was solely responsible for conceptualization, methodology, illustrative evaluation design, writing, and review of this article. [Adjust to a contribution statement covering all listed authors before deposit if additional authors are added.]

Acknowledgments. [Insert acknowledgments here if applicable before deposit.]

REFERENCES

1. Atlassian. (2022). Connect Azure DevOps to Jira. Atlassian Support. Retrieved from https://support.atlassian.com/jira-cloud-administration/docs/integrate-azure-devops-with-jira/
2. AXELOS. (2019). ITIL 4: Release management practice guide. AXELOS Limited.
3. DORA. (2019). Accelerate: State of DevOps 2019. DevOps Research and Assessment. Retrieved from https://dora.dev/research/2019/dora-report/2019-dora-accelerate-state-of-devops-report.pdf
4. Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The science of lean software and DevOps: Building and scaling high performing technology organizations. IT Revolution Press.
5. Gotel, O. C. Z., & Finkelstein, A. C. W. (1994). An analysis of the requirements traceability problem. Proceedings of the IEEE International Conference on Requirements Engineering, 1994. Retrieved from http://www0.cs.ucl.ac.uk/staff/A.Finkelstein/papers/rtprob.pdf
6. Humble, J., & Farley, D. (2010). Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison Wesley.
7. ISACA. (2018). COBIT 2019 framework: Introduction and methodology. ISACA.
8. Microsoft. (2022a). Link work items to objects. Azure Boards documentation, Microsoft Learn. Retrieved from https://learn.microsoft.com/en-us/azure/devops/boards/backlogs/add-link
9. Microsoft. (2022b). Drive Git development from work items. Azure Boards documentation, Microsoft Learn. Retrieved from https://learn.microsoft.com/en-us/azure/devops/boards/backlogs/connect-work-items-to-git-dev-ops
10. Microsoft. (2022c). Automate work item completion with pull requests. Azure Boards documentation, Microsoft Learn. Retrieved from https://learn.microsoft.com/en-us/azure/devops/boards/work-items/auto-complete-work-items-pull-requests
11. Microsoft. (2022d). YAML schema reference for Azure Pipelines. Microsoft Learn. Retrieved from https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/
12. Schwaber, K., & Sutherland, J. (2020). The Scrum Guide: The definitive guide to Scrum: The rules of the game. scrumguides.org. Retrieved from https://scrumguides.org/
13. Stahl, D., & Hallen, J. (2016). Continuous integration and delivery traceability in industry: Needs and practices. IEEE. Retrieved from https://ieeexplore.ieee.org/document/7592779/