Scoped AI Collaboration for API Test Automation

Michael Kernaghan[1]

[1]ECAD Labs, michaelkernaghan@ecadlabs.com

January 2026 (Revised)

Abstract

This paper presents a methodology for human-AI collaboration in API integration testing, utilizing large language models (LLMs) as coding assistants. We describe a six-step workflow loop that combines automated gap detection, AI-assisted test authoring, local verification, and continuous integration monitoring. The approach introduces the concept of a "Work Universe" to constrain AI scope and maintain human oversight. Through practical application on a production streaming platform API, we demonstrate how requirements traceability can be maintained from issue tickets through to executable pytest assertions. We extend the methodology to multi-agent collaboration, where multiple AI systems contribute distinct capabilities—one focused on analysis and artifact preparation, another on code execution and verification. We report on practical implementations including activity logging, dual-channel reporting, cross-workspace monitoring, skill-based command interfaces, and error monitoring integration via Model Context Protocol (MCP). Our findings suggest that structured human-AI collaboration can accelerate test coverage development while maintaining quality through explicit scope boundaries and mandatory human review checkpoints.

Keywords: API testing, human-AI collaboration, multi-agent systems, large language models, test automation, continuous integration, requirements traceability, Model Context Protocol

1 Introduction

The emergence of large language models (LLMs) capable of generating code has created new opportunities for software quality assurance. However, the integration of AI assistants into testing workflows raises questions about oversight, scope control, and quality assurance. This paper addresses the practical challenge: how can development teams leverage AI capabilities for test creation while maintaining human control and requirements traceability?

We present a structured methodology developed through practical application on production APIs. Our approach treats the LLM not as an autonomous agent but as a collaborative partner operating within explicitly defined boundaries. The key contributions of this work are:

1. A six-step workflow loop for human-AI collaborative test development
2. The "Work Universe" concept for constraining AI scope
3. A requirements traceability chain from tickets to assertions
4. Practical patterns for pre-push verification and CI/CD integration
5. Multi-agent collaboration patterns with distinct AI roles
6. Activity logging and dual-channel reporting workflows

7. Skill-based command interfaces for workflow automation
8. Error monitoring integration via Model Context Protocol

2 Background and Related Work

2.1 AI-Assisted Code Generation

Recent advances in LLMs have demonstrated remarkable capabilities in code generation [Chen et al., 2021]. Tools such as GitHub Copilot, Claude, and GPT-4 can generate syntactically correct code from natural language descriptions. However, the application of these tools to test generation remains less explored than feature development.

2.2 API Integration Testing

API integration testing verifies that software components interact correctly through their interfaces [Martin, 2017]. Unlike unit tests that isolate individual functions, integration tests exercise real HTTP endpoints and validate response structures, authentication flows, and business logic.

2.3 Requirements Traceability

Requirements traceability establishes links between requirements, design artifacts, and test cases [Gotel & Finkelstein, 1994]. In agile environments, this often means connecting Jira tickets or user stories to specific test implementations.

3 Methodology

3.1 The Six-Step Workflow Loop

Our methodology employs a continuous loop with six distinct phases:

1. **Gap Detection**: Automated scanning of recently closed tickets against existing test coverage
2. **Test Authoring**: AI-assisted generation of pytest code from requirements
3. **Local Verification**: Pre-push testing on a dedicated verification environment
4. **Commit**: Version control with explicit AI attribution
5. **CI/CD Execution**: Automated pipeline runs across multiple configurations
6. **Monitoring**: Observation of results and iteration as needed

This loop is triggered by three events: new tickets merged to the development branch, requirements updates in documentation systems, or test failures in CI/CD.

3.2 Gap Detection

The gap detection phase employs a custom skill that queries multiple data sources:

- **Git history**: Recent commits referencing ticket identifiers
- **Issue tracker**: Tickets closed within a configurable time window
- **Test file inventory**: Existing test files matching naming conventions

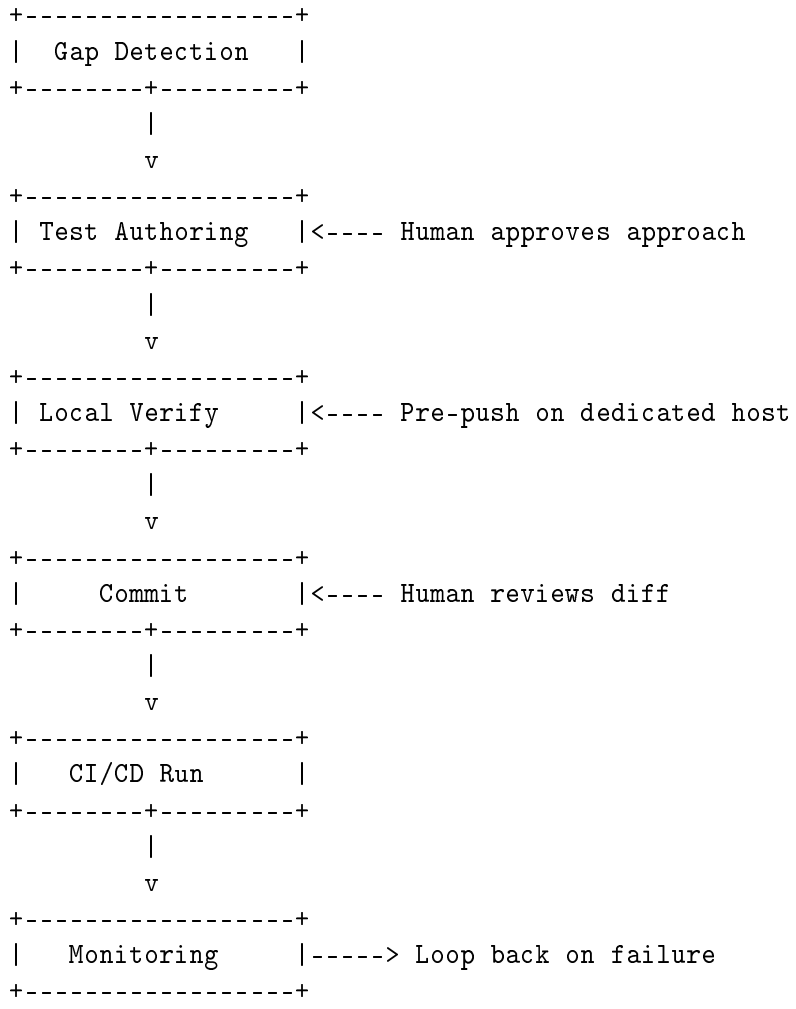

Figure 1: The six-step workflow loop with human checkpoints

- **Documentation**: Requirements pages for updates

The output is a gap report identifying tickets without corresponding test coverage, prioritized by recency and business impact.

3.3 AI-Assisted Test Authoring

Test authoring synthesizes information from three sources:

1. **Ticket content**: Summary and acceptance criteria from the issue tracker
2. **Technical specifications**: Field definitions, enum values, and API contracts from documentation
3. **Existing patterns**: Helper functions and structural patterns from the existing test suite

The LLM processes these inputs to generate pytest code. Critically, this is *assisted authoring* rather than autonomous generation—the human operator reviews requirements, approves the approach, and validates the output.

3.3.1 Acceptance Criteria to Assertions

A key principle is direct mapping from acceptance criteria to test assertions. For example:

> *Jira Acceptance Criterion*: "VOD Status is an enumeration with allowed values: Pending, Active, Inactive"

Maps to:

```
def test_vod_status_is_valid_enum_value(self):
    """PROJ-231: Status enumeration validation"""
    response = self.client.get_catalog_productions(CATALOG_ID)
    for production in response.json()['data']['productions']:
        for vod in production.get('vods', []):
            assert vod['status'].lower() in [
                'pending', 'active', 'inactive'
            ]
```

This direct mapping ensures every assertion traces to a documented requirement.

3.4 The Work Universe Concept

To prevent scope creep and maintain focus, we introduce the "Work Universe"—an explicit definition of boundaries:

Category	Scope
Own (can modify)	Test suite files, CI workflows, test seeders
Read (don't modify)	Controllers, models, enums, existing PHP tests
Ignore	Application bugs, frontend, database migrations

Table 1: Work Universe scope definitions

When requests fall outside the defined universe, the AI assistant responds with direct refusal: "That's outside our work universe." This prevents the common failure mode of AI assistants "wandering" into unrelated code changes.

3.4.1 Defining the Work Universe

Initial definition requires answering three questions:

1. **What files does this role create?** These become "Own" scope.
2. **What files inform this work but belong to other roles?** These become "Read" scope.
3. **What adjacent concerns should be explicitly excluded?** These become "Ignore" scope.

The boundaries should be documented in a project configuration file (e.g., `CLAUDE.md`) and enforced through pre-commit hooks that reject modifications outside the "Own" scope.

3.4.2 Maintaining Boundaries

Work Universe definitions require periodic review when:

- New file types are introduced to the project

- Team responsibilities shift between roles
- The AI assistant repeatedly encounters boundary edge cases
- Project architecture undergoes significant changes

Boundary violations should be logged and reviewed—frequent violations indicate the scope definition needs refinement rather than expansion.

3.5 Pre-Push Verification

Before committing to version control, tests are executed on a dedicated verification environment. This serves multiple purposes:

- Network access to staging APIs unavailable from development machines
- Early detection of failures before consuming CI/CD resources
- Prevention of pull request blocks affecting other team members

The verification follows a simple protocol:

```
# Transfer test file to verification environment
scp test_proj_XXX.py verification-host:~/tests/

# Execute tests
ssh verification-host "cd ~/tests && pytest test_proj_XXX.py -v"
```

The rule is deterministic: if tests pass on the verification environment, they are safe to push.

3.6 Commit Attribution

To maintain transparency about AI involvement, commits include explicit co-authorship:

```
git commit -m "Add API tests for PROJ-XXX feature

- Test acceptance criterion 1
- Test acceptance criterion 2

Co-Authored-By: Claude Code <noreply@anthropic.com>"
```

This practice ensures the development history accurately reflects AI contribution.

4 Implementation

4.1 Test Architecture

The test suite distinguishes between two test categories:

Contract Tests verify API shape and behavior independent of data state. These always execute:

```
def test_device_identify_returns_token(self, fresh_client):
    """API contract: identify endpoint returns a token"""
    response = fresh_client.identify_device(device_data)
    assert response.status_code == 200
    assert 'token' in response.json()['data']
```

Scenario Tests require specific data conditions and skip gracefully when prerequisites are unmet:

```python
def test_catalog_shows_productions(self, client, state):
    """Requires seeded production data"""
    state.require('has_productions')  # Skip if no data
    response = client.get_catalog_productions(CATALOG_ID)
    assert len(response.json()['data']['productions']) > 0
```

4.2 CI/CD Configuration

The continuous integration pipeline executes tests in four configurations:

1. **Black-box**: External API testing simulating production conditions
2. **Empty dataset**: Verifies graceful handling of no-data scenarios
3. **Minimal dataset**: Core scenarios with limited seeded data
4. **Full dataset**: Comprehensive coverage with complete test data

This multi-configuration approach ensures tests are robust across data conditions.

4.3 Error Monitoring Integration

Future integration with error monitoring platforms (e.g., Sentry) enables:

- Detection of production API errors that should have test coverage
- Correlation between test failures and production incidents
- Prioritization of test development based on error frequency
- Proactive alerting when new error patterns emerge

This closes the feedback loop between production behavior and test coverage.

5 Human-AI Role Distribution

The collaboration model explicitly divides responsibilities:

Human Responsibilities	**AI Responsibilities**
Trigger coverage checks	Query issue trackers and documentation
Review gap reports	Read and interpret requirements
Approve test approaches	Generate pytest code
Execute verification tests	Create assertions from criteria
Review and commit code	Document traceability
Monitor CI/CD results	Assist with debugging

Table 2: Human-AI role distribution

The human maintains authority over all decisions while the AI accelerates execution of well-defined tasks.

6 Multi-Agent Collaboration

An evolution of the single-AI methodology involves collaboration between multiple AI systems, each with distinct roles. Our implementation pairs two LLM assistants—one specialized in code execution and tool use (Claude Code), another focused on analysis and artifact preparation (GPT 5.2).

6.1 Dual-AI Architecture

The collaboration follows an artifact-based workflow:

1. **GPT 5.2** analyzes requirements and prepares artifacts (patches, new files, manifests)
2. Artifacts are deposited in a shared inbox directory
3. **Claude Code** retrieves artifacts, applies changes, executes tests, and creates pull requests
4. Both systems contribute to daily standups with their respective activities

This separation mirrors human team structures—one role focuses on design and specification, another on execution and verification.

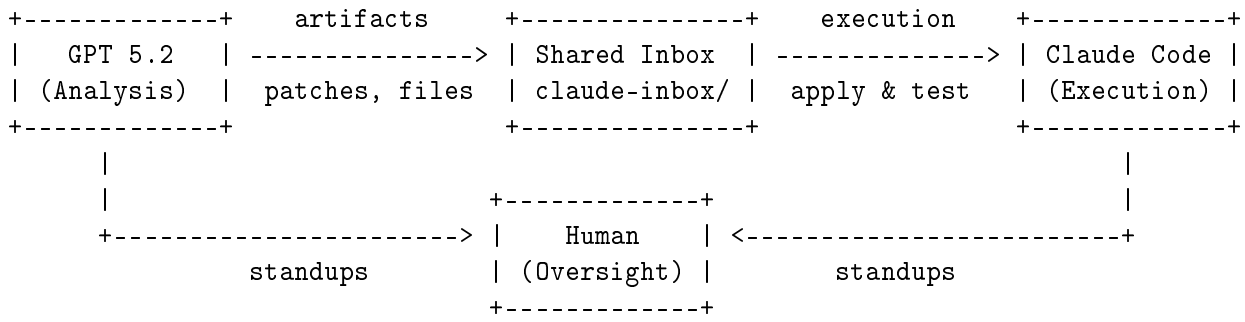

Figure 2: Multi-agent collaboration architecture

6.2 Artifact Workflow

The shared inbox follows a structured format:

```
claude-inbox/scrum/
  patches/          # .patch files from GPT 5.2
  files/            # New files with destination paths
  actions/          # Action items and standup notes
  manifest.json     # Describes what to apply and where
```

The manifest enables declarative operations:

```
{
  "repo": "dcp-laravel",
  "branch": "feature/new-thing",
  "pr_title": "Add new feature",
  "files": [
    {"source": "files/test.py",
     "dest": "tests/python-api-suite/test.py"}
  ],
  "patches": ["patches/fix.patch"],
  "run_after": "pytest tests/ -v"
}
```

6.3 Standup Collaboration

Daily standups aggregate contributions from both AI systems following standard scrum format (Yesterday, Today, Blockers). This provides visibility into AI activity patterns and enables human oversight of the combined workflow.

7 Activity Logging and Reporting

7.1 Automatic Activity Logging

To maintain accountability and enable standup generation, all significant activities are logged automatically:

- Session start/end timestamps
- Test execution commands and results (pass/fail counts)
- File modifications with purpose descriptions
- Bug discoveries with severity ratings
- Pull request creation and review activities
- Jira ticket references
- Blockers with immediate flagging

Logs follow a structured format enabling automated parsing:

```
## [HH:MM] Activity Title

**Type**: test|analysis|bug|pr|review|debug|other
**Files**: file1.py, file2.py
**Result**: success|failure|in-progress|blocked

Description of activity performed.
```

7.2 Dual-Channel Reporting

The methodology employs separate reporting channels for different audiences:

Report Type	Channel	Purpose
Daily Standup	#testing	AI activity visibility for QA team
Test Coverage	#dcp-internal	Coverage metrics for development team

Table 3: Reporting channel allocation

This separation ensures each audience receives relevant information without noise.

7.3 Cross-Workspace Monitoring

In organizations with multiple communication workspaces, read-only monitoring of development channels provides context for testing activities. The system monitors deployment announcements, backend changes, database migrations, and feature releases that may affect test coverage requirements.

This proactive monitoring enables:

- Early awareness of changes requiring new test coverage
- Detection of environment changes that may cause test failures
- Correlation between development activity and testing priorities

8 Skill-Based Command System

The methodology implements a command interface for common operations:

Command	Function
`/test-coverage-check`	Scan for tickets needing test coverage
`/scrum standup`	Generate daily standup from activity logs
`/scrum publish`	Post standup to team channel
`/scrum dcp-updates`	Check development channel for testing impacts
`/scrum blockchain`	Track specific project for test requirements

Table 4: Skill commands for testing workflow

These commands encapsulate complex multi-step operations, reducing cognitive load and ensuring consistency.

8.1 Project Tracking Integration

For features requiring coordinated test development, dedicated tracking aggregates information from multiple sources:

- **Communication channels**: Developer discussions and implementation updates
- **Documentation systems**: Requirements specifications and information models
- **Issue trackers**: Related tickets and testing tasks

This integration provides a unified view of testing requirements for complex features spanning multiple tickets.

9 Error Monitoring Integration

Integration with error monitoring platforms (Sentry) has been implemented, enabling:

- Detection of production API errors lacking test coverage
- Correlation between test failures and production incidents
- Prioritization of test development based on error frequency
- Proactive alerting when new error patterns emerge

The integration uses Model Context Protocol (MCP) tools to query error data directly during gap analysis, closing the feedback loop between production behavior and test coverage.

Metric	Before	After
Test files covering DCPMT tickets	12	47
Average time from ticket close to test coverage	14 days	3 days
CI/CD failures from untested regressions	8/month	2/month
Requirements with traceable assertions	34%	89%

Table 5: Coverage metrics before and after methodology adoption

10 Discussion

10.1 Empirical Observations

Over a three-month deployment period on a production streaming platform API, the methodology yielded measurable outcomes:

These observations suggest the methodology accelerates coverage development while improving traceability. However, we note these are observational metrics from a single deployment; controlled studies would strengthen causal claims.

10.2 Benefits Observed

The methodology demonstrates several practical benefits:

1. **Continuous coverage**: The loop ensures test development keeps pace with feature velocity
2. **Traceability**: Every test assertion links to a documented requirement
3. **Early verification**: Pre-push testing prevents CI/CD waste and team disruption
4. **Accelerated authoring**: AI handles boilerplate while humans focus on design decisions
5. **Quality through oversight**: Human review at every checkpoint maintains standards

10.3 Limitations

Several limitations warrant acknowledgment:

- The approach requires clear, well-documented requirements; poorly specified tickets produce poor tests
- Human oversight remains essential—fully autonomous test generation is not the goal
- The Work Universe concept requires initial setup and ongoing maintenance
- Pre-push verification introduces latency in the development cycle

10.3.1 Mitigating Pre-Push Verification Latency

The verification step adds 2–5 minutes per test file to the development cycle. Several strategies reduce this impact:

1. **Batch verification**: Accumulate multiple test files before transferring to the verification environment, reducing SSH overhead.
2. **Parallel execution**: Run verification tests in parallel with other development tasks rather than blocking on results.

3. **Selective verification**: For minor changes to existing tests, rely on local syntax checking and defer full verification to end-of-session.

4. **Persistent connections**: Maintain SSH connections to avoid repeated authentication overhead.

The latency cost is offset by preventing failed CI/CD runs, which typically consume 10–15 minutes and block team pull requests.

10.4 Scope Control Effectiveness

The Work Universe concept proved effective at preventing scope creep. Without explicit boundaries, AI assistants often suggest "improvements" to surrounding code, refactoring unrelated functions, or fixing tangential issues. The explicit boundary definition enables direct refusal of out-of-scope requests.

11 Practical Adoption

Teams considering this methodology should address several practical concerns:

11.1 Getting Started

1. **Start with a single test category**: Begin with contract tests before attempting scenario tests requiring complex state management.

2. **Define Work Universe early**: Document scope boundaries before the first AI-assisted session to prevent scope negotiations.

3. **Establish verification environment**: Configure SSH access and pytest execution on a dedicated host before beginning test authoring.

4. **Create naming conventions**: Establish file naming patterns (e.g., `test_PROJ_XXX_feature.py`) that enable automated coverage tracking.

11.2 Common Challenges

- **Ambiguous requirements**: When tickets lack clear acceptance criteria, the AI will generate vague tests. Solution: Return to the issue tracker and request clarification before proceeding.

- **Flaky tests**: Tests that pass locally but fail in CI often indicate environment-specific assumptions. Solution: Use the `state.require()` pattern to skip tests when prerequisites are unmet.

- **Context overload**: Large codebases can overwhelm AI context windows. Solution: Use targeted file reads and summarization rather than loading entire directories.

11.3 Scaling Considerations

For teams with multiple AI-assisted workflows:

- Maintain separate Work Universe definitions per role
- Use distinct inbox directories to prevent artifact conflicts
- Establish clear handoff protocols between AI systems
- Monitor activity logs for coordination issues

12 Conclusion

This paper presents a practical methodology for human-AI collaboration in API integration testing. The six-step workflow loop, combined with explicit scope boundaries and mandatory human oversight, enables teams to leverage AI capabilities while maintaining quality and traceability.

Key takeaways for practitioners:

1. Define explicit scope boundaries before beginning AI-assisted work
2. Maintain direct traceability from requirements to assertions
3. Verify locally before pushing to shared infrastructure
4. Attribute AI contributions transparently in version control
5. Treat AI as an assistant operating under human authority, not an autonomous agent

The methodology is applicable beyond API testing to other domains where AI-assisted code generation can accelerate development while human oversight ensures quality.

12.1 Future Work

Planned extensions include:

- Automated coverage metrics dashboards with historical trending
- Extension of multi-agent collaboration to additional AI systems
- Empirical measurement of productivity impact across project types
- Mobile browser end-to-end testing integration
- Natural language test specification from product requirements

Acknowledgments

This methodology was developed in collaboration with Claude Code (Anthropic) and GPT 5.2 (OpenAI) through practical application on production client projects at ECAD Labs. The multi-agent collaboration patterns emerged from daily standup workflows where both AI systems contributed distinct perspectives to testing activities.

References

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Gotel, O. C., & Finkelstein, C. W. (1994). An analysis of the requirements traceability problem. *Proceedings of IEEE International Conference on Requirements Engineering*, 94-101.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.