SPEC-DOC-004: Testing & Quality Assurance Documentation

Status: Pending **Priority**: P1 (Medium) **Estimated Effort**: 12-16 hours **Target Audience**: Contributors, QA engineers **Created**: 2025-11-17

Objectives

Document the complete testing and QA infrastructure: 1. Testing strategy (coverage goals: 40%+ target, currently 42-48%) 2. Test infrastructure (MockMcpManager, fixtures, tarpaulin) 3. Unit testing guide (patterns, examples, mocking) 4. Integration testing (workflow tests, cross-module) 5. E2E testing (pipeline validation, tmux automation) 6. Property-based testing (proptest, edge cases) 7. CI/CD integration (GitHub workflows, pre-commit hooks) 8. Performance testing (benchmarking, profiling)

Scope

In Scope

- Testing strategy and coverage targets (42-48% achieved, targeting 40%+)
- Test infrastructure (MockMcpManager implementation, fixtures)
- Unit testing patterns and examples
- Integration testing approach (604 tests total, 100% pass rate)
- E2E testing with tmux automation
- Property-based testing with proptest
- CI/CD workflows (.github/workflows/)
- Pre-commit/pre-push hooks
- Performance testing and benchmarking

- Test organization (per-module, integration tests)

Out of Scope

- Writing new tests (implementation work)
- Internal testing policy details (covered in testing-policy.md)
- Spec-kit functional testing (covered in SPEC-DOC-003)

Deliverables

1. **content/testing-strategy.md** - Coverage goals, module targets
2. **content/test-infrastructure.md** - MockMcpManager, fixtures, tools
3. **content/unit-testing-guide.md** - Patterns, examples, mocking
4. **content/integration-testing-guide.md** - Workflow tests, cross-module
5. **content/e2e-testing-guide.md** - Pipeline validation, tmux
6. **content/property-testing-guide.md** - Proptest usage, edge cases
7. **content/ci-cd-integration.md** - GitHub workflows, hooks
8. **content/performance-testing.md** - Benchmarking, profiling

---

Success Criteria

□ Testing strategy clearly documented
□ MockMcpManager usage fully explained
□ Unit test patterns demonstrated with examples
□ Integration test approach documented
□ CI/CD workflow explained
□ All 604 existing tests referenced

Related SPECs

- SPEC-DOC-000 (Master)
- SPEC-DOC-002 (Core Architecture - testing architecture)
- SPEC-DOC-005 (Development - running tests locally)

Status: Structure defined, content pending

CI/CD Integration

Comprehensive guide to CI/CD integration and automated testing.

Overview

CI/CD Testing Strategy: Automated testing at every stage (pre-commit, pre-push, CI, release)

Goals: - Fast feedback (<5s pre-commit, <2min pre-push) - Comprehensive coverage (all tests in CI) - Prevent regressions - Maintain code quality

Current Status: - Pre-commit hooks: 100% adoption - CI pipeline: GitHub Actions - Test execution: ~10-15 minutes - Pass rate: 100%

Testing Stages

1. Pre-Commit (Local) - <5s

Purpose: Fast policy checks before commit

Location: `.githooks/pre-commit`

What it runs:

```
# Only runs if spec_kit modules modified
# Check 1: Storage policy
bash scripts/validate_storage_policy.sh

# Check 2: Tag schema
bash scripts/validate_tag_schema.sh
```

Execution Time: <5s

Bypass (emergencies only):

```
git commit --no-verify
```

---

2. Pre-Push (Local) - ~2-5 min

Purpose: Compile and lint checks before push

Triggered: Before `git push`

What it runs:

```
# Format check
cargo fmt --all -- --check

# Linting
cargo clippy --workspace --all-targets --all-features -- -D warnings

# Build (all features)
cargo build --workspace --all-features

# Optional: Targeted test compilation
cargo test --workspace --no-run
```

Execution Time: ~2-5 minutes

Bypass (emergencies only):

```
PREPUSH_FAST=0 git push
```

---

3. CI/CD (GitHub Actions) - ~10-15 min

Purpose: Complete testing and release

Triggered: Push to `main`, pull requests

Location: `.github/workflows/release.yml`

Jobs: 1. **Preflight Tests** (Linux fast E2E) 2. **Determine Version** (semantic versioning) 3. **Build** (Linux, macOS, Windows) 4. **Test** (all tests, all platforms) 5. **Release** (npm publish)

GitHub Actions Workflows

Preflight Tests Job

Purpose: Fast integration tests before full build matrix

Platform: Ubuntu 24.04

Steps:

```yaml
jobs:
  preflight-tests:
    name: Preflight Tests (Linux fast E2E)
    runs-on: ubuntu-24.04
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Install Rust (1.90)
        run: |
          rustup set profile minimal
          rustup toolchain install 1.90.0 --profile minimal
          rustup default 1.90.0

      - name: Setup Rust Cache
        uses: Swatinem/rust-cache@v2
        with:
          prefix-key: v5-rust
          shared-key: codex-preflight-1.90
          workspaces: codex-rs -> target
          cache-targets: true
          cache-on-failure: true

      - name: Build (fast profile)
        run: ./build-fast.sh

      - name: Curated tests + CLI smokes
        run: bash scripts/ci-tests.sh
```

What it tests:

```bash
# scripts/ci-tests.sh

# Curated integration tests
cargo test -p codex-login --test all -q
cargo test -p codex-chatgpt --test all -q
cargo test -p codex-apply-patch --test all -q
cargo test -p codex-execpolicy --tests -q
cargo test -p mcp-types --tests -q

# CLI smoke tests
./codex-rs/target/dev-fast/code --version
./codex-rs/target/dev-fast/code completion bash
./codex-rs/target/dev-fast/code doctor
```

Execution Time: ~3-5 minutes

Benefits: - ✅ Fast feedback (before full matrix) - ✅ Catches common errors early - ✅ Tests critical integration points - ✅ Validates CLI functionality

---

Build Matrix Job

Purpose: Build and test on all platforms

Platforms: - Linux (Ubuntu 24.04, x64 + arm64) - macOS (latest, x64 + arm64) - Windows (latest, x64)

Rust Versions: - Stable (1.90) - Beta (optional)

Steps:

```yaml
jobs:
  build:
    strategy:
      matrix:
        os: [ubuntu-24.04, macos-latest, windows-latest]
        rust: [1.90.0]
    runs-on: ${{ matrix.os }}
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Install Rust
        run: rustup toolchain install ${{ matrix.rust }}

      - name: Build
        run: cargo build --workspace --all-features

      - name: Test
        run: cargo test --workspace --all-features
```

Execution Time: ~8-12 minutes per platform

Release Job

Purpose: Publish to npm after successful tests

Triggers: - Push to `main` - All tests pass

Steps: 1. Determine version (semantic versioning) 2. Build binaries (all platforms) 3. Publish to npm (`@just-every/code`)

Packages Published: - `@just-every/code` (main package) - `@just-every/code-darwin-arm64` - `@just-every/code-darwin-x64` - `@just-every/code-linux-x64-musl` - `@just-every/code-linux-arm64-musl` - `@just-every/code-win32-x64`

Pre-Commit Hook

Installation

One-time setup:

```
bash scripts/setup-hooks.sh
```

Verifies:

```
git config core.hooksPath
# Should output: .githooks
```

What It Checks

File: `.githooks/pre-commit`

```
#!/bin/bash
# Pre-commit hook for policy compliance

# Only run if spec_kit modules modified
SPEC_KIT_CHANGES=$(git diff --cached --name-only | grep "spec_kit" || true)

if [ -z "$SPEC_KIT_CHANGES" ]; then
    # No spec_kit changes, skip policy checks
    exit 0
fi

echo "🔍 Running policy compliance checks (spec_kit modified)..."

# Check 1: Storage policy
if ! bash scripts/validate_storage_policy.sh; then
    echo "❌ Storage policy violation detected"
    exit 1
fi

# Check 2: Tag schema
if ! bash scripts/validate_tag_schema.sh; then
    echo "❌ Tag schema violation detected"
    exit 1
fi

echo "✅ Policy compliance checks passed"
exit 0
```

Checks: 1. **Storage policy**: Ensures local-memory usage compliant (MEMORY-POLICY.md) 2. **Tag schema**: Validates tag namespacing and naming

Performance: <5s (only runs for spec_kit changes)

Bypass Pre-Commit (Emergencies Only)

```
# Skip hook (use sparingly)
git commit --no-verify -m "Emergency fix"
```

When to bypass: - Critical production hotfix - Hook infrastructure broken - Reviewing/reverting broken commits

When NOT to bypass: - Avoiding policy violations (fix the code instead) - Convenience (hooks are fast) - Regular workflow

---

CI Test Script

Location

File: `scripts/ci-tests.sh`

Purpose: Fast integration tests for CI

What It Tests

```
#!/usr/bin/env bash
set -euo pipefail

echo "[ci-tests] Running curated integration tests..."
pushd codex-rs >/dev/null

# Login integration tests
cargo test -p codex-login --test all -q

# ChatGPT integration tests
cargo test -p codex-chatgpt --test all -q

# Apply patch integration tests
cargo test -p codex-apply-patch --test all -q

# Execution policy tests
cargo test -p codex-execpolicy --tests -q

# MCP types tests
cargo test -p mcp-types --tests -q

popd >/dev/null

echo "[ci-tests] CLI smokes with host binary..."
BIN=./codex-rs/target/dev-fast/code

# Smoke tests
"${BIN}" --version >/dev/null
"${BIN}" completion bash >/dev/null
"${BIN}" doctor >/dev/null || true

echo "[ci-tests] Done."
```

Why Curated Tests?

Full test suite: 604 tests, ~15 minutes

Curated subset: ~150 tests, ~3-5 minutes

Selection Criteria: - ✅ Integration tests (cross-module) - ✅ E2E tests (complete workflows) - ✅ Critical paths (login, apply, MCP) - ❌ Unit tests (fast, covered by local dev) - ❌ Property tests (slow, covered by weekly runs)

Benefits: - ✅ Fast feedback (3-5 min vs 15 min) - ✅ High signal (integration tests find real bugs) - ✅ CI efficiency (parallel preflight + full tests)

Local Testing Before Push

Recommended Workflow

Step 1: Run affected tests (iterative development):

```
cd codex-rs

# Test specific module you changed
cargo test -p codex-tui --lib

# Test specific file
cargo test -p codex-tui spec_kit::clarify_native
```

Step 2: Run full test suite (before committing):

```
cd codex-rs
cargo test --workspace --all-features
```

Step 3: Check format and lint (before committing):

```
cd codex-rs
cargo fmt --all -- --check
cargo clippy --workspace --all-targets --all-features -- -D warnings
```

Step 4: Commit (pre-commit hook runs automatically):

```
git add .
git commit -m "feat(tui): add clarify native checks"
# Hook runs: storage policy, tag schema (<5s)
```

Step 5: Push (pre-push hook runs automatically, optional):

```
git push
# Hook runs: fmt, clippy, build (~2-5min)
```

Fast Iteration Loop

For rapid development:

```
# 1. Make changes
vim codex-rs/tui/src/chatwidget/spec_kit/clarify_native.rs

# 2. Test just this module (fast)
cd codex-rs
```

```
cargo test -p codex-tui clarify_native -- --nocapture

# 3. If tests pass, run clippy on this crate
cargo clippy -p codex-tui -- -D warnings

# 4. Commit (hook runs policy checks)
git add codex-rs/tui
git commit -m "fix(clarify): improve ambiguity detection"

# 5. Push later after multiple commits
git push
```

Execution Time: - Module tests: ~5-10s - Clippy: ~15-30s - Commit: <5s (hook) - **Total**: ~30-50s per iteration

---

Code Coverage Integration

Local Coverage Measurement

Tool: cargo-tarpaulin or cargo-llvm-cov

Install:

```
cargo install cargo-tarpaulin
# or
cargo install cargo-llvm-cov
```

Usage:

```
cd codex-rs

# Generate coverage report
cargo tarpaulin --workspace --all-features --out Html

# Open report
open target/tarpaulin/index.html
```

CI Coverage (Future)

GitHub Actions (not yet implemented):

```
jobs:
  coverage:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install tarpaulin
        run: cargo install cargo-tarpaulin

      - name: Run coverage
        run: cargo tarpaulin --workspace --all-features --out Xml

      - name: Upload to Codecov
        uses: codecov/codecov-action@v3
        with:
          files: ./cobertura.xml
      - name: Comment PR with coverage
        uses: codecov/codecov-action@v3
```

Benefits (when implemented): - ✅ Track coverage trends - ✅ Fail PR if coverage drops >5% - ✅ Visualize coverage in PRs

---

Best Practices

DO

✅ **Run tests locally before pushing**:

```
# Always test before pushing
cargo test --workspace --all-features

# Push after tests pass
git push
```

✅ **Fix CI failures immediately**:

```
# CI failed? Fix it now, not later
git pull
cargo test --workspace
# Fix failures
git commit -m "fix(ci): resolve test failures"
git push
```

✅ **Keep CI green**: - Main branch should always pass tests - Revert breaking commits if fix takes >1 hour - Document known flaky tests

---

✅ **Use caching effectively**:

```
# GitHub Actions caching
- uses: Swatinem/rust-cache@v2
  with:
    prefix-key: v5-rust
    shared-key: codex-preflight-1.90
```

✅ **Run curated tests in CI** (fast feedback):

```
# Preflight: curated subset (3-5 min)
bash scripts/ci-tests.sh

# Full matrix: all tests (10-15 min)
cargo test --workspace --all-features
```

DON’T

❌ **Skip pre-commit hooks routinely**:

```
# Bad: Habitual bypassing
git commit --no-verify  # ❌ Don't make this a habit
```

❌ **Push without testing**:

```bash
# Bad: Push untested code
git commit -m "quick fix"
git push  # ❌ No local testing
```

❌ **Ignore CI failures**:

```
# Bad: "CI is always red anyway"
# ❌ Fix CI or revert
```

❌ **Commit broken tests**:

```rust
# Bad: Disable failing tests instead of fixing
#[test]
#[ignore]  // ❌ Don't ignore, fix!
fn test_that_fails() { }
```

❌ **Let coverage drop**:

```
# Bad: Coverage drops from 45% to 30%
# ❌ Add tests, don't delete them
```

Troubleshooting

Pre-Commit Hook Not Running

Symptom: Commits succeed without running hook

Fix:

```bash
# Check git config
git config core.hooksPath
# Should output: .githooks

# If not set, run setup
bash scripts/setup-hooks.sh
```

CI Timeout

Symptom: CI job times out after 60 minutes

Causes: - Infinite loop in test - Deadlock in concurrent test - Slow property test (PROPTEST_CASES too high)

Fix:

```bash
# Find slow tests locally
cargo test --workspace -- --nocapture --test-threads=1

# Reduce property test cases
PROPTEST_CASES=100 cargo test --test property_based_tests
```

Flaky Tests

Symptom: Test passes locally, fails in CI (or vice versa)

Common Causes: - Race conditions (concurrent tests) - Hardcoded paths (use TempDir) - Network dependencies (use mocks) - Time-dependent tests (use fixed timestamps)

Fix:

```bash
# Run test multiple times locally
for i in {1..100}; do
    cargo test test_flaky_name || break
done

# If it fails, debug with single thread
cargo test test_flaky_name -- --test-threads=1 --nocapture
```

Build Cache Corruption

Symptom: Build fails in CI with cryptic errors, passes locally

Fix (GitHub Actions):

```yaml
# Clear cache by changing cache key
- uses: Swatinem/rust-cache@v2
  with:
    prefix-key: v6-rust  # Increment version
```

Summary

CI/CD Testing Stages:

1. **Pre-Commit** (<5s): Policy checks (storage, tags)
2. **Pre-Push** (2-5min): Format, clippy, build
3. **Preflight Tests** (3-5min): Curated integration tests
4. **Full CI** (10-15min): All tests, all platforms
5. **Release** (auto): Publish on main after tests pass

Tools: - ✅ GitHub Actions (CI/CD) - ✅ Rust Cache (faster builds) - ✅ Git Hooks (pre-commit, pre-push) - ✅ cargo-tarpaulin (coverage)

Best Practices: - ✅ Test locally before pushing - ✅ Keep CI green (100% pass rate) - ✅ Fast feedback (curated tests in preflight) - ✅ Fix failures immediately - ✅ Use caching (Rust Cache)

Next Steps: - Performance Testing - Benchmarks and profiling - Testing Strategy - Overall testing approach - Test Infrastructure - MockMcpManager, fixtures

---

References: - GitHub Actions: `.github/workflows/release.yml` - Pre-commit hook: `.githooks/pre-commit` - CI test script: `scripts/ci-tests.sh` - Setup hooks: `scripts/setup-hooks.sh`

End-to-End Testing Guide

Comprehensive guide to end-to-end testing of complete user workflows.

Overview

End-to-End (E2E) Testing Philosophy: Test complete user workflows from start to finish, simulating real-world usage

Goals: - Validate critical user journeys - Test system integration (TUI + backend + database + MCP) - Verify error recovery and degradation - Ensure configuration hot-reload works

Current Status: - ~24 E2E tests (4% of total) - 100% pass rate - Average execution time: ~10-60s per test - Categories: Pipeline automation, quality checkpoints, tmux sessions, config hot-reload

---

E2E Test Categories

Pipeline Automation Tests

Purpose: Test complete `/speckit.auto` pipeline (Plan → Tasks → Implement → Validate → Audit → Unlock)

Location: `codex-rs/tui/tests/spec_auto_e2e.rs`

Coverage: - Pipeline state machine (initialization, transitions, resume) - Quality checkpoint integration (PrePlanning, PostPlan, PostTasks) - Stage progression (all 6 stages) - Error handling and recovery

---

Quality Checkpoint Tests

Purpose: Test quality gates at critical pipeline points

Checkpoints: - **PrePlanning** (BeforeSpecify): Clarify ambiguities before plan - **PostPlan** (AfterSpecify): Checklist quality scoring after plan - **PostTasks** (AfterTasks): Analyze consistency after tasks

Coverage: - Checkpoint triggering - Modification tracking - Escalation logic - Human intervention

---

Tmux Session Tests

Purpose: Test tmux integration for long-running operations

Location: `codex-rs/evidence/tmux-automation/`

Coverage: - Session creation and lifecycle - Agent spawning in background - Session termination - Evidence collection

---

Config Hot-Reload Tests

Purpose: Test configuration changes without restart

Location: `codex-rs/tui/tests/config_reload_integration_tests.rs`

Coverage: - Config file watching - Hot-reload triggers - Provider switching - <100ms latency (p95)

Pipeline E2E Tests

Test Structure

Standard Pattern:

```rust
#[test]
fn test_spec_auto_state_initialization() {
    // 1. Create initial state
    let state = SpecAutoState::new(
        "SPEC-TEST-001".to_string(),
        "Test automation".to_string(),
        SpecStage::Plan,
        None,  // HAL mode
    );

    // 2. Assert initial conditions
    assert_eq!(state.spec_id, "SPEC-TEST-001");
    assert_eq!(state.goal, "Test automation");
    assert_eq!(state.current_index, 0);
    assert_eq!(state.stages.len(), 6);
    assert_eq!(state.current_stage(), Some(SpecStage::Plan));
    assert!(state.quality_gates_enabled);
    assert!(state.completed_checkpoints.is_empty());
}
```

Pattern 1: Pipeline Initialization

Example: spec_auto_e2e.rs:20

```rust
#[test]
fn test_spec_auto_state_initialization() {
    let state = SpecAutoState::new(
        "SPEC-TEST-001".to_string(),
        "Test automation".to_string(),
        SpecStage::Plan,
        None,
    );

    // Verify initial state
    assert_eq!(state.spec_id, "SPEC-TEST-001");
    assert_eq!(state.goal, "Test automation");
    assert_eq!(state.current_index, 0);
    assert_eq!(state.current_stage(), Some(SpecStage::Plan));

    // Verify stages
    assert_eq!(state.stages.len(), 6);
    let expected = vec![
        SpecStage::Plan,
        SpecStage::Tasks,
        SpecStage::Implement,
        SpecStage::Validate,
        SpecStage::Audit,
        SpecStage::Unlock,
    ];
    assert_eq!(state.stages, expected);

    // Verify quality gates
    assert!(state.quality_gates_enabled);
    assert!(state.completed_checkpoints.is_empty());
}
```

What This Tests: - ✅ State initialization - ✅ Stage ordering (Plan → Tasks → Implement → Validate → Audit → Unlock) - ✅ Quality gates enabled by default - ✅ Checkpoint tracking initialized

---

Pattern 2: Pipeline Stage Progression

```
#[test]
fn test_pipeline_full_progression() {
    let mut state = SpecAutoState::new(
        "SPEC-TEST-002".to_string(),
        "Full pipeline test".to_string(),
        SpecStage::Plan,
        None,
    );

    // ==================== PLAN STAGE ====================
    assert_eq!(state.current_stage(), Some(SpecStage::Plan));
    assert_eq!(state.current_index, 0);

    // Simulate plan completion
    state.current_index += 1;

    // ==================== TASKS STAGE ====================
    assert_eq!(state.current_stage(), Some(SpecStage::Tasks));
    assert_eq!(state.current_index, 1);

    state.current_index += 1;

    // ==================== IMPLEMENT STAGE ====================
    assert_eq!(state.current_stage(), Some(SpecStage::Implement));
    assert_eq!(state.current_index, 2);

    state.current_index += 1;

    // ==================== VALIDATE STAGE ====================
    assert_eq!(state.current_stage(), Some(SpecStage::Validate));
    assert_eq!(state.current_index, 3);

    state.current_index += 1;

    // ==================== AUDIT STAGE ====================
    assert_eq!(state.current_stage(), Some(SpecStage::Audit));
    assert_eq!(state.current_index, 4);
    state.current_index += 1;

    // ==================== UNLOCK STAGE ====================
    assert_eq!(state.current_stage(), Some(SpecStage::Unlock));
    assert_eq!(state.current_index, 5);

    // ==================== COMPLETION ====================
    state.current_index += 1;
    assert_eq!(state.current_stage(), None); // Pipeline complete
}
```

What This Tests: - ✅ All 6 stages execute in order - ✅ Index advances correctly - ✅ State transitions deterministically - ✅ Pipeline completion (stage = None)

Pattern 3: Resume from Middle Stage

```
#[test]
fn test_resume_from_tasks_stage() {
    // Start from Tasks (not Plan)
    let state = SpecAutoState::new(
        "SPEC-TEST-003".to_string(),
        "Resume test".to_string(),
        SpecStage::Tasks,  // Resume from Tasks
        None,
    );

    // Verify resume point
    assert_eq!(state.current_index, 1); // Tasks is index 1
    assert_eq!(state.current_stage(), Some(SpecStage::Tasks));

    // Verify can still progress
    let mut state = state;
    state.current_index += 1;
    assert_eq!(state.current_stage(), Some(SpecStage::Implement));
}
```

What This Tests: - ✅ Pipeline can resume from any stage - ✅ Index calculated correctly for resume - ✅ Progression continues normally

---

Quality Checkpoint E2E Tests

Pattern 1: Checkpoint Tracking

```
#[test]
fn test_quality_checkpoints_track_completion() {
    let mut state = SpecAutoState::new(
        "SPEC-TEST-006".to_string(),
        "Checkpoint tracking".to_string(),
        SpecStage::Plan,
        None,
    );

    // Initially no checkpoints completed
    assert!(state.completed_checkpoints.is_empty());
```

```
            // ==================== PRE-PLANNING CHECKPOINT
====================

            // Simulate PrePlanning checkpoint (Clarify)

state.completed_checkpoints.insert(QualityCheckpoint::PrePlanning);

            assert!
(state.completed_checkpoints.contains(&QualityCheckpoint::PrePlanning));

            assert!
(!state.completed_checkpoints.contains(&QualityCheckpoint::PostPlan));

            assert_eq!(state.completed_checkpoints.len(), 1);

            // ==================== POST-PLAN CHECKPOINT
====================

            // Simulate PostPlan checkpoint (Checklist)
            state.completed_checkpoints.insert(QualityCheckpoint::PostPlan);

            assert!
(state.completed_checkpoints.contains(&QualityCheckpoint::PrePlanning));

            assert!
(state.completed_checkpoints.contains(&QualityCheckpoint::PostPlan));

            assert_eq!(state.completed_checkpoints.len(), 2);

            // ==================== POST-TASKS CHECKPOINT
====================

            // Simulate PostTasks checkpoint (Analyze)

state.completed_checkpoints.insert(QualityCheckpoint::PostTasks);

            assert_eq!(state.completed_checkpoints.len(), 3);
            assert!
(state.completed_checkpoints.contains(&QualityCheckpoint::PostTasks));

        }
```

What This Tests: - ✅ Checkpoint completion tracked - ✅ Multiple checkpoints can coexist - ✅ No duplicate checkpoints (Set semantics)

Pattern 2: Quality Modifications Tracking

```
        #[test]
        fn test_quality_modifications_tracked() {
            let mut state = SpecAutoState::new(
                "SPEC-TEST-007".to_string(),
                "Modification tracking".to_string(),
                SpecStage::Plan,
                None,
            );

            // Initially no modifications
            assert!(state.quality_modifications.is_empty());
```

```
    // ==================== PREPLANNING MODIFICATIONS ====================

    // User fixes ambiguities in spec.md
    state.quality_modifications.push("spec.md".to_string());

    assert_eq!(state.quality_modifications.len(), 1);
    assert!(state.quality_modifications.contains(&"spec.md".to_string()));

    // ==================== POSTPLAN MODIFICATIONS ====================

    // User improves plan.md after checklist
    state.quality_modifications.push("plan.md".to_string());

    assert_eq!(state.quality_modifications.len(), 2);
    assert!(state.quality_modifications.contains(&"plan.md".to_string()));

    // ==================== POSTTASKS MODIFICATIONS ====================

    // User fixes tasks.md after analyze
    state.quality_modifications.push("tasks.md".to_string());

    assert_eq!(state.quality_modifications.len(), 3);
}
```

What This Tests: - ✅ Modifications tracked across checkpoints - ✅ Multiple files can be modified - ✅ Modification history preserved

---

Pattern 3: Quality Gates Can Be Disabled

```
#[test]
fn test_quality_gates_can_be_disabled() {
    let state = SpecAutoState::with_quality_gates(
        "SPEC-TEST-008".to_string(),
        "No quality gates".to_string(),
        SpecStage::Plan,
        None,
        false,  // Disable quality gates
    );

    // Verify quality gates disabled
    assert!(!state.quality_gates_enabled);

    // Pipeline should skip all checkpoints
    // (checkpoint logic would check quality_gates_enabled flag)
}
```

What This Tests: - ✅ Quality gates can be disabled - ✅ Flag persists in state - ✅ Pipeline can run without checkpoints

---

Real-World E2E Tests

Pattern 1: Apply Command E2E

Example: apply_command_e2e.rs:78

```
        #[tokio::test]
        async fn test_apply_command_creates_fibonacci_file() {
            // ==================== SETUP: TEMP GIT REPO
====================

            let temp_repo = create_temp_git_repo()
                .await
                .expect("Failed to create temp git repo");
            let repo_path = temp_repo.path();

            // ==================== LOAD TASK FIXTURE ====================

            let task_response = mock_get_task_with_fixture()
                .await
                .expect("Failed to load fixture");

            // ==================== EXECUTE: APPLY DIFF ====================

            apply_diff_from_task(task_response,
Some(repo_path.to_path_buf()))
                .await
                .expect("Failed to apply diff from task");

            // ==================== VERIFY: FILE CREATED
====================

            let fibonacci_path = repo_path.join("scripts/fibonacci.js");
            assert!(fibonacci_path.exists(), "fibonacci.js was not
created");

            // ==================== VERIFY: FILE CONTENTS
====================

            let contents = std::fs::read_to_string(&fibonacci_path)
                .expect("Failed to read fibonacci.js");

            assert!(
                contents.contains("function fibonacci(n)"),
                "fibonacci.js doesn't contain expected function"
            );
        }
```

Helper: Create Temp Git Repo:

```
        async fn create_temp_git_repo() -> anyhow::Result<TempDir> {
            let temp_dir = TempDir::new()?;
            let repo_path = temp_dir.path();
            let envs = vec![
                ("GIT_CONFIG_GLOBAL", "/dev/null"),
                ("GIT_CONFIG_NOSYSTEM", "1"),
            ];

            // Initialize git repo
            Command::new("git")
                .envs(envs.clone())
                .args(["init"])
                .current_dir(repo_path)
                .output()
                .await?;

            // Configure user
            Command::new("git")
                .envs(envs.clone())
                .args(["config", "user.email", "test@example.com"])
                .current_dir(repo_path)
                .output()
                .await?;

            Command::new("git")
                .envs(envs.clone())
                .args(["config", "user.name", "Test User"])
                .current_dir(repo_path)
                .output()
                .await?;

            // Create initial commit
            std::fs::write(repo_path.join("README.md"), "# Test Repo\n")?;

            Command::new("git")
                .envs(envs.clone())
                .args(["add", "README.md"])
                .current_dir(repo_path)
                .output()
                .await?;

            Command::new("git")
                .envs(envs.clone())
                .args(["commit", "-m", "Initial commit"])
                .current_dir(repo_path)
                .output()
                .await?;

            Ok(temp_dir)
        }
```

What This Tests: - ✅ Complete apply command workflow - ✅ Git integration (temp repo, commits) - ✅ File creation and modification - ✅ Task fixture loading

---

Pattern 2: Login Flow E2E

Example: login_server_e2e.rs:79

```
        #[tokio::test]
        async fn end_to_end_login_flow_persists_auth_json() -> Result<()> {
            // ==================== SETUP: MOCK OAuth ISSUER
====================

            let (issuer_addr, issuer_handle) = start_mock_issuer();
            let issuer = format!("http://{}:{}", issuer_addr.ip(),
issuer_addr.port());

            // ==================== SETUP: TEMP CODEX HOME
====================

            let tmp = tempdir()?;
            let codex_home = tmp.path().to_path_buf();

            // Seed auth.json with stale data (should be overwritten)
            let stale_auth = serde_json::json!({
                "OPENAI_API_KEY": "sk-stale",
                "tokens": {
                    "id_token": "stale.header.payload",
                    "access_token": "stale-access",
                    "refresh_token": "stale-refresh",
                }
            });
            std::fs::write(
                codex_home.join("auth.json"),
                serde_json::to_string_pretty(&stale_auth)?,
            )?;

            // ==================== EXECUTE: LOGIN FLOW ====================

            let options = ServerOptions {
                issuer: issuer.clone(),
                redirect_uri: "http://localhost:8080/callback".to_string(),
                codex_home: codex_home.clone(),
                // ... other options
            };

            run_login_server(options).await?;

            // ==================== VERIFY: AUTH.JSON UPDATED
====================

            let updated_auth =
std::fs::read_to_string(codex_home.join("auth.json"))?;
            let auth_data: serde_json::Value =
serde_json::from_str(&updated_auth)?;

            // Verify tokens refreshed
            assert_ne!(auth_data["tokens"]["access_token"], "stale-access");
            assert_eq!(auth_data["tokens"]["access_token"], "access-123");

            // ==================== CLEANUP: SHUTDOWN MOCK
====================

            drop(issuer_handle);

            Ok(())
        }
```

Helper: Start Mock OAuth Issuer:

```
        fn start_mock_issuer() -> (SocketAddr, thread::JoinHandle<()>) {
            let listener = TcpListener::bind(("127.0.0.1", 0)).unwrap();
            let addr = listener.local_addr().unwrap();
            let server = tiny_http::Server::from_listener(listener,
None).unwrap();

            let handle = thread::spawn(move || {
                while let Ok(mut req) = server.recv() {
                    let url = req.url().to_string();
                    if url.starts_with("/oauth/token") {
                        // Build minimal JWT
                        let payload = serde_json::json!({
                            "email": "user@example.com",
                            "https://api.openai.com/auth": {
                                "chatgpt_plan_type": "pro",
                            }
                        });

                        let id_token = create_jwt(&payload);

                        let tokens = serde_json::json!({
                            "id_token": id_token,
                            "access_token": "access-123",
                            "refresh_token": "refresh-123",
                        });

                        let resp = tiny_http::Response::from_data(
                            serde_json::to_vec(&tokens).unwrap()
                        );
                        let _ = req.respond(resp);
                    }
                }
            });

            (addr, handle)
        }
```

What This Tests: - ✅ Complete login flow - ✅ OAuth integration (mock issuer) - ✅ Token persistence (auth.json) - ✅ Stale token replacement

---

E2E Test Setup Patterns

Pattern 1: Temp Git Repository

```
        async fn create_temp_git_repo() -> anyhow::Result<TempDir> {
            let temp_dir = TempDir::new()?;
            let repo_path = temp_dir.path();

            // Disable global git config (isolation)
            let envs = vec![
                ("GIT_CONFIG_GLOBAL", "/dev/null"),
                ("GIT_CONFIG_NOSYSTEM", "1"),
            ];

            // Initialize repo
            run_git_command(repo_path, &envs, &["init"]).await?;

            // Configure user (required for commits)
            run_git_command(repo_path, &envs, &["config", "user.email",
"test@example.com"]).await?;
            run_git_command(repo_path, &envs, &["config", "user.name", "Test
User"]).await?;

            // Create initial commit
            std::fs::write(repo_path.join("README.md"), "# Test\n")?;
            run_git_command(repo_path, &envs, &["add", "."]).await?;
            run_git_command(repo_path, &envs, &["commit", "-m", "Initial
commit"]).await?;
```

```
    Ok(temp_dir)
}

async fn run_git_command(
    repo_path: &Path,
    envs: &[(&str, &str)],
    args: &[&str],
) -> anyhow::Result<()> {
    let output = Command::new("git")
        .envs(envs.iter().copied())
        .args(args)
        .current_dir(repo_path)
        .output()
        .await?;

    if !output.status.success() {
        anyhow::bail!(
            "Git command failed: {}",
            String::from_utf8_lossy(&output.stderr)
        );
    }

    Ok(())
}
```

Benefits: - ✅ Isolated from global git config - ✅ Auto-cleanup (TempDir) - ✅ Reusable helper functions

---

Pattern 2: Mock HTTP Server

```
fn start_mock_server() -> (SocketAddr, thread::JoinHandle<()>) {
    // Bind to random port
    let listener = TcpListener::bind(("127.0.0.1", 0)).unwrap();
    let addr = listener.local_addr().unwrap();
    let server = tiny_http::Server::from_listener(listener,
None).unwrap();

    let handle = thread::spawn(move || {
        while let Ok(req) = server.recv() {
            let url = req.url().to_string();

            let response = match url.as_str() {
                "/api/v1/endpoint" => {
                    serde_json::json!({"status": "ok"})
                }
                _ => {
                    serde_json::json!({"error": "not found"})
                }
            };

            let resp = tiny_http::Response::from_data(
                serde_json::to_vec(&response).unwrap()
            );
            let _ = req.respond(resp);
        }
    });

    (addr, handle)
}
```

```
#[tokio::test]
async fn test_with_mock_server() {
    let (addr, _handle) = start_mock_server();
    let base_url = format!("http://{}:{}", addr.ip(), addr.port());

    // Test code using base_url...
}
```

Benefits: - ✅ No external dependencies - ✅ Deterministic responses - ✅ Fast (no network)

Pattern 3: Fixture Loading

```
async fn load_fixture<T: serde::de::DeserializeOwned>(name: &str) -> anyhow::Result<T> {
    let fixture_path = Path::new(env!("CARGO_MANIFEST_DIR"))
        .join("tests/fixtures")
        .join(format!("{}.json", name));

    let contents = std::fs::read_to_string(fixture_path)?;
    let data: T = serde_json::from_str(&contents)?;

    Ok(data)
}

#[tokio::test]
async fn test_with_fixture() {
    let task: GetTaskResponse = load_fixture("task_turn_fixture")
        .await
        .expect("Failed to load fixture");

    // Use task...
}
```

Benefits: - ✅ Realistic test data - ✅ Reusable across tests - ✅ Version controlled

---

Best Practices

DO

✅ **Test complete user workflows**:

```
// Good: Tests entire pipeline
#[test]
fn test_speckit_auto_full_pipeline() {
    // Create state
    // Run plan
    // Run tasks
    // ... all 6 stages
    // Verify completion
}
```

✅ **Use realistic test data**:

```rust
// Good: Load from fixture
let task = load_fixture("real_task_response").await?;

// Bad: Minimal mock data
let task = GetTaskResponse { id: "1", content: "test" };
```

✅ **Verify side effects**:

```rust
// Verify file created
assert!(fibonacci_path.exists());

// Verify contents correct
let contents = std::fs::read_to_string(&fibonacci_path)?;
assert!(contents.contains("function fibonacci"));

// Verify git commit
let log = run_git(&["log", "--oneline"]).await?;
assert!(log.contains("Add fibonacci.js"));
```

✅ **Test error recovery**:

```rust
#[tokio::test]
async fn test_pipeline_recovers_from_mcp_failure() {
    // Simulate MCP failure
    mock_mcp.fail_next_request();

    // Run pipeline
    let result = run_pipeline().await;

    // Verify fallback succeeded
    assert!(result.is_ok());
    assert!(result.unwrap().degraded);
}
```

✅ **Clean up resources**:

```rust
#[tokio::test]
async fn test_with_cleanup() {
    let temp_dir = TempDir::new()?;
    let (_addr, handle) = start_mock_server();

    // Test logic...

    // Cleanup
    drop(handle);  // Shutdown mock server
    drop(temp_dir);  // Delete temp files

    Ok(())
}
```

DON’T

❌ **Test too many workflows in one test**:

```rust
// Bad: Tests multiple workflows (hard to debug)
#[test]
fn test_all_commands() {
    test_apply_command();
    test_login_flow();
    test_config_reload();
    test_tmux_session();
    // ... 500 lines
}
```

✗ Rely on external services:

```
// Bad: Depends on real OpenAI API
#[tokio::test]
async fn test_real_api() {
    let response =
reqwest::get("https://api.openai.com/v1/models").await?;
    // ✗ Flaky, slow, costs money
}

// Good: Use mock server
#[tokio::test]
async fn test_with_mock() {
    let (addr, _handle) = start_mock_server();
    let base_url = format!("http://{}", addr);
    // ✅ Fast, deterministic, free
}
```

✗ Skip verification:

```
// Bad: No assertions
#[tokio::test]
async fn test_pipeline() {
    run_pipeline().await?;
    // ✗ No verification
}

// Good: Verify outcomes
#[tokio::test]
async fn test_pipeline() {
    let result = run_pipeline().await?;
    assert_eq!(result.stages_completed, 6);
    assert!(result.plan_file.exists());
}
```

Running E2E Tests

Run All E2E Tests

```
cd codex-rs
cargo test --test '*_e2e'
```

Runs: - `spec_auto_e2e.rs` - `apply_command_e2e.rs` - `login_server_e2e.rs`

Run Specific E2E Test

```
cargo test --test spec_auto_e2e test_spec_auto_state_initialization
```

Run with Verbose Output

```
cargo test --test spec_auto_e2e -- --nocapture --test-threads=1
```

Why `--test-threads=1`: - E2E tests may conflict (ports, files) - Single-threaded ensures isolation

---

Summary

E2E Testing Best Practices:

1. **Complete Workflows**: Test from start to finish
2. **Realistic Data**: Use fixtures from real usage
3. **Isolation**: Temp dirs, mock servers, disable global config
4. **Verification**: Check files, state, side effects
5. **Error Recovery**: Test fallback and degradation
6. **Cleanup**: Auto-cleanup with TempDir, handle drops

Test Categories: - ✅ Pipeline automation (/speckit.auto, 6 stages) - ✅ Quality checkpoints (PrePlanning, PostPlan, PostTasks) - ✅ Real-world workflows (apply command, login flow) - ✅ Configuration hot-reload

Key Patterns: - ✅ Temp git repositories (isolated, auto-cleanup) - ✅ Mock HTTP servers (tiny_http, deterministic) - ✅ Fixture loading (realistic test data) - ✅ State machine validation (initialization, progression, resume)

Next Steps: - Property Testing Guide - Generative invariant testing - CI/CD Integration - Automated testing pipeline - Performance Testing - Benchmarks and profiling

---

References: - Pipeline E2E: `codex-rs/tui/tests/spec_auto_e2e.rs` - Apply command: `codex-rs/chatgpt/tests/suite/apply_command_e2e.rs` - Login flow: `codex-rs/login/tests/suite/login_server_e2e.rs`

Integration Testing Guide

Comprehensive guide to integration testing across modules.

Overview

Integration Testing Philosophy: Test multiple modules working together in realistic workflows

Goals: - Verify module interactions - Test cross-cutting concerns (error recovery, state persistence) - Validate end-to-end workflows - Ensure evidence integrity

Current Status: - ~200 integration tests (33% of total) - 100% pass rate - Average execution time: ~3-5s per test - Categories: W01-W15 (workflows), E01-E15 (errors), S01-S10 (state), Q01-Q10 (quality), C01-C10 (concurrent)

Integration Test Categories

W01-W15: Workflow Integration Tests

Purpose: Test complete stage workflows across modules

Flow: Handler → Consensus → Evidence → Guardrail → State

Location: `codex-rs/tui/tests/workflow_integration_tests.rs`

Coverage: - W01-W05: Individual stage workflows (Plan, Tasks, Implement, Validate, Audit) - W06-W10: Multi-stage pipelines - W11-W15: Quality gate integration

---

E01-E15: Error Recovery Integration Tests

Purpose: Test error propagation and recovery across modules

Flow: Error → Retry → Fallback → Recovery → Evidence

Location: `codex-rs/tui/tests/error_recovery_integration_tests.rs`

Coverage: - E01-E05: Consensus and MCP failures - E06-E10: Guardrail validation errors - E11-E15: State corruption and recovery

---

S01-S10: State Persistence Integration Tests

Purpose: Test state coordination with evidence storage

Flow: State Change → Evidence Write → Load from Disk → Reconstruct

Location: `codex-rs/tui/tests/state_persistence_integration_tests.rs`

Coverage: - S01-S05: State serialization and reconstruction - S06-S10: Pipeline interrupt and resume

---

Q01-Q10: Quality Gate Integration Tests

Purpose: Test quality gate orchestration across modules

Flow: Quality Gate → Native Checks → Consensus → Escalation → Guardrail

Location: `codex-rs/tui/tests/quality_flow_integration_tests.rs`

Coverage: - Q01-Q05: BeforeSpecify and AfterSpecify gates - Q06-Q10: AfterTasks gate and consensus validation

---

C01-C10: Concurrent Operations Integration Tests

Purpose: Test concurrent stage execution and evidence locking

Flow: Parallel Stages → Lock Acquisition → Evidence Writes → Lock Release

Location: `codex-rs/tui/tests/concurrent_operations_integration_tests.rs`

Coverage: - C01-C05: Parallel consensus collection - C06-C10: Evidence write contention

---

Test Structure

Standard Integration Test Pattern

```rust
#[test]
fn w01_plan_stage_complete_workflow() {
    // 1. Setup: Create test context
    let ctx = IntegrationTestContext::new("SPEC-W01-001").unwrap();

    // 2. Arrange: Prepare filesystem (PRD, spec files)
    ctx.write_prd("test-feature", "# Test Feature\nBuild a test feature")
        .unwrap();
    ctx.write_spec("test-feature", "# Specification\nDetailed spec")
        .unwrap();

    // 3. Arrange: Create initial state
    let mut state = StateBuilder::new("SPEC-W01-001")
        .with_goal("Build test feature")
        .starting_at(SpecStage::Plan)
        .build();

    // 4. Act: Simulate module interactions
    // Write mock consensus artifacts (simulating consensus module output)
    let consensus_file = ctx
        .consensus_dir()
        .join("spec-plan_2025-10-19T12_00_00Z_gemini.json");
    std::fs::write(
        &consensus_file,
        json!({
            "agent": "gemini",
            "content": "Plan consensus output",
            "timestamp": "2025-10-19T12:00:00Z"
        })
        .to_string(),
    )
    .unwrap();

    // Write mock guardrail telemetry (simulating guardrail module output)
    let guardrail_file = ctx
        .commands_dir()
        .join("spec-plan_2025-10-19T12_00_00Z.json");
    std::fs::write(
        &guardrail_file,
        json!({
            "schemaVersion": 1,
            "baseline": {"status": "passed"},
            "tool": {"status": "passed"},
        })
        .to_string(),
    )
    .unwrap();

    // 5. Assert: Verify evidence
    let verifier = EvidenceVerifier::new(&ctx);
    assert!(verifier.assert_structure_valid());
    assert!(ctx.assert_consensus_exists(SpecStage::Plan, "gemini"));
    assert!(ctx.assert_guardrail_telemetry_exists(SpecStage::Plan));

    // 6. Assert: Verify state transitions
    state.current_index += 1;
    assert_eq!(state.current_stage(), Some(SpecStage::Tasks));

    // 7. Assert: Verify artifact counts
    assert_eq!(ctx.count_consensus_files(), 1);
    assert_eq!(ctx.count_guardrail_files(), 1);
}
```

Workflow Integration Tests

Pattern 1: Individual Stage Workflow

Example: W01 - Plan Stage Complete Workflow

Test (workflow_integration_tests.rs:22):

```
#[test]
fn w01_plan_stage_complete_workflow() {
    let ctx = IntegrationTestContext::new("SPEC-W01-001").unwrap();

    // Arrange: Create PRD and spec
    ctx.write_prd("test-feature", "# Test Feature\nBuild a test
feature")
        .unwrap();
    ctx.write_spec("test-feature", "# Specification\nDetailed spec")
        .unwrap();

    // Arrange: Initial state
    let mut state = StateBuilder::new("SPEC-W01-001")
        .with_goal("Build test feature")
        .starting_at(SpecStage::Plan)
        .build();

    assert_eq!(state.current_stage(), Some(SpecStage::Plan));

    // Act: Simulate consensus module output
    let consensus_file = ctx
        .consensus_dir()
        .join("spec-plan_2025-10-19T12_00_00Z_gemini.json");
    std::fs::write(
        &consensus_file,
        json!({
            "agent": "gemini",
            "content": "Plan consensus output",
        })
```

```
                    .to_string(),
            )
            .unwrap();

            // Act: Simulate guardrail module output
            let guardrail_file = ctx
                .commands_dir()
                .join("spec-plan_2025-10-19T12_00_00Z.json");
            std::fs::write(
                &guardrail_file,
                json!({"schemaVersion": 1, "baseline": {"status":
"passed"}})
                    .to_string(),
            )
            .unwrap();

            // Assert: Verify evidence
            assert!(ctx.assert_consensus_exists(SpecStage::Plan, "gemini"));
            assert!(ctx.assert_guardrail_telemetry_exists(SpecStage::Plan));

            // Assert: Verify state advancement
            state.current_index += 1;
            assert_eq!(state.current_stage(), Some(SpecStage::Tasks));
        }
```

Pattern 2: Multi-Stage Pipeline

Example: W06 - Plan → Tasks Pipeline

```
        #[test]
        fn w06_plan_tasks_pipeline() {
            let ctx = IntegrationTestContext::new("SPEC-W06-001").unwrap();

            // Arrange: Initial setup
            ctx.write_prd("multi-stage", "# Multi-stage Test").unwrap();
            let mut state = StateBuilder::new("SPEC-W06-001")
                .starting_at(SpecStage::Plan)
                .build();

            // ==================== PLAN STAGE ====================

            // Act: Plan stage consensus
            let plan_consensus = ctx
                .consensus_dir()
                .join("spec-plan_2025-10-19T10_00_00Z_gemini.json");
            std::fs::write(
                &plan_consensus,
                json!({"agent": "gemini", "stage": "plan", "content": "Plan
output"})
                    .to_string(),
            )
            .unwrap();

            // Assert: Plan evidence exists
            assert!(ctx.assert_consensus_exists(SpecStage::Plan, "gemini"));

            // Advance to Tasks
            state.current_index += 1;
            assert_eq!(state.current_stage(), Some(SpecStage::Tasks));
```

```
    // ==================== TASKS STAGE ====================

    // Act: Tasks stage consensus
    let tasks_consensus = ctx
        .consensus_dir()
        .join("spec-tasks_2025-10-19T10_05_00Z_claude.json");
    std::fs::write(
        &tasks_consensus,
        json!({"agent": "claude", "stage": "tasks", "content": "Task list"})
            .to_string(),
    )
    .unwrap();

    // Assert: Tasks evidence exists (accumulated, not replaced)
    assert!(ctx.assert_consensus_exists(SpecStage::Plan, "gemini"));
    assert!(ctx.assert_consensus_exists(SpecStage::Tasks, "claude"));
    assert_eq!(ctx.count_consensus_files(), 2);

    // Advance to Implement
    state.current_index += 1;
    assert_eq!(state.current_stage(), Some(SpecStage::Implement));
}
```

Key Points: - ✅ Evidence accumulates across stages (not replaced) - ✅ State advances sequentially - ✅ Each stage verified independently

---

Pattern 3: Quality Gate Integration

Example: W11 - BeforeSpecify Quality Gate

```
#[test]
fn w11_before_specify_quality_gate_workflow() {
    let ctx = IntegrationTestContext::new("SPEC-W11-001").unwrap();

    // Arrange: Create PRD with known ambiguities
    ctx.write_prd(
        "test",
        r#"
# PRD
## Requirements
- R1: System should be fast
- R2: Must handle TBD authentication
        "#,
    )
    .unwrap();

    let mut state = StateBuilder::new("SPEC-W11-001")
        .quality_gates(true)
        .starting_at(SpecStage::Plan)
        .build();

    // Act: Simulate quality gate execution (Clarify)
    let quality_gate_result = ctx
        .commands_dir()
        .join("quality-gate-clarify_2025-10-19T10_00_00Z.json");
    std::fs::write(
                &quality_gate_result,
                json!({
                    "gate": "BeforeSpecify",
                    "checks": ["clarify"],
                    "results": {
                        "ambiguities": [
                            {"pattern": "should", "severity": "Important"},
                            {"pattern": "TBD", "severity": "Critical"}
                        ]
                    },
                    "verdict": "escalate",  // Critical issues found
                    "escalation_reason": "2 ambiguities found (1 critical)"
                })
                .to_string(),
            )
            .unwrap();

            // Assert: Quality gate escalated
            let content =
std::fs::read_to_string(&quality_gate_result).unwrap();
            let data: serde_json::Value =
serde_json::from_str(&content).unwrap();
            assert_eq!(data["verdict"], "escalate");
            assert!(data["results"]["ambiguities"]
                .as_array()
                .unwrap()
                .len() > 0);

            // State remains at Plan (doesn't advance on escalation)
            assert_eq!(state.current_stage(), Some(SpecStage::Plan));
        }
```

Key Points: - ✅ Quality gate runs before stage - ✅ Escalation prevents advancement - ✅ Evidence records escalation reason

---

Error Recovery Integration Tests

Pattern 1: Consensus Failure → Retry → Recovery

Example: E01 - Consensus Failure with Retry

Test (error_recovery_integration_tests.rs:23):

```
        #[test]
        fn
e01_consensus_failure_handler_retry_evidence_cleanup_state_reset() {
            let ctx = IntegrationTestContext::new("SPEC-E01-001").unwrap();

            let mut state = StateBuilder::new("SPEC-E01-001")
                .starting_at(SpecStage::Plan)
                .build();

            // ==================== ATTEMPT 1: FAILURE ====================

            // Act: Write failed consensus (empty result)
            let failed_consensus = ctx
                .consensus_dir()
                .join("spec-plan_2025-10-
19T10_00_00Z_gemini_attempt1.json");
            std::fs::write(
                &failed_consensus,
                json!({
                    "agent": "gemini",
                    "stage": "plan",
                    "status": "failed",
                    "error": "Empty consensus result",
                    "attempt": 1,
                })
                .to_string(),
            )
            .unwrap();

            // Assert: Failed attempt recorded
            assert!(failed_consensus.exists());

            // ==================== RETRY: CLEANUP ====================

            // Simulate retry: cleanup failed evidence
            std::fs::remove_file(&failed_consensus).unwrap();
            assert!(!failed_consensus.exists());

            // ==================== ATTEMPT 2: SUCCESS ====================

            // Act: Retry with enhanced prompt
            let success_consensus = ctx
                .consensus_dir()
                .join("spec-plan_2025-10-
19T10_05_00Z_gemini_attempt2.json");
            std::fs::write(
                &success_consensus,
                json!({
                    "agent": "gemini",
                    "stage": "plan",
                    "status": "success",
                    "content": "Enhanced prompt successful",
                    "attempt": 2,
                    "retry_reason": "empty_result",
                })
                .to_string(),
            )
            .unwrap();

            // Assert: Retry succeeded
            assert!(success_consensus.exists());
            assert_eq!(ctx.count_consensus_files(), 1); // Only successful
attempt remains

            // Assert: State advances after successful retry
            state.current_index += 1;
            assert_eq!(state.current_stage(), Some(SpecStage::Tasks));

            // Assert: Evidence shows retry metadata
            let content =
std::fs::read_to_string(&success_consensus).unwrap();
            assert!(content.contains("retry_reason"));
            assert!(content.contains("attempt"));
        }
```

Key Points: - ✅ Failed attempt recorded as evidence - ✅ Retry cleanup removes failed attempt - ✅ Success includes retry metadata - ✅ State advances only on success

---

Pattern 2: MCP Failure → Fallback → Recovery

Example: E02 - MCP Timeout with File Fallback

```
        #[test]
        fn e02_mcp_failure_fallback_to_file_evidence_records_fallback() {
            let ctx = IntegrationTestContext::new("SPEC-E02-001").unwrap();

            // ==================== MCP FAILURE ====================

            // Write fallback marker evidence (MCP failed, using file
fallback)
            let fallback_evidence = ctx
                .consensus_dir()
                .join("spec-plan_mcp_fallback_2025-10-19T10_00_00Z.json");
            std::fs::write(
                &fallback_evidence,
                json!({
                    "fallback_mode": "file_based",
                    "mcp_error": "Timeout after 60s",
                    "fallback_timestamp": "2025-10-19T10:00:00Z"
                })
                .to_string(),
            )
            .unwrap();

            // Assert: Fallback recorded
            assert!(fallback_evidence.exists());

            // ==================== FILE-BASED CONSENSUS
====================

            // Act: Write consensus from file-based fallback
            let file_consensus = ctx
                .consensus_dir()
                .join("spec-plan_2025-10-19T10_00_00Z_file_based.json");
            std::fs::write(
                &file_consensus,
                json!({
                    "source": "file_based_fallback",
                    "content": "Consensus from local files",
                    "degraded": true
                })
                .to_string(),
            )
            .unwrap();

            // Assert: File-based consensus succeeded
            assert!(file_consensus.exists());
            assert_eq!(ctx.count_consensus_files(), 2); // Fallback marker +
consensus

            // Assert: Degraded flag set
            let content = std::fs::read_to_string(&file_consensus).unwrap();
            assert!(content.contains("\"degraded\":true"));
}
```

Key Points: - ✅ MCP failure recorded as fallback evidence - ✅ File-based fallback produces consensus - ✅ Degraded flag indicates fallback mode - ✅ Multiple evidence files coexist

---

State Persistence Integration Tests

Pattern 1: State Serialization → Load → Reconstruct

Example: S01 - State Persistence and Reconstruction

Test (state_persistence_integration_tests.rs:18):

```
#[test]
fn s01_state_change_evidence_write_load_from_disk_reconstruct() {
    let ctx = IntegrationTestContext::new("SPEC-S01-001").unwrap();
    let state = StateBuilder::new("SPEC-S01-001")
        .starting_at(SpecStage::Plan)
        .build();

    // ==================== SERIALIZE STATE ====================

    // Act: Write state to evidence
    let state_file =
ctx.commands_dir().join("spec_auto_state.json");
    std::fs::write(
        &state_file,
        json!({
            "spec_id": state.spec_id,
            "current_index": state.current_index,
            "quality_gates_enabled": state.quality_gates_enabled,
        })
        .to_string(),
    )
    .unwrap();

    // ==================== LOAD AND RECONSTRUCT
====================

    // Act: Load from disk and verify reconstruction
    let loaded = std::fs::read_to_string(&state_file).unwrap();
    let data: serde_json::Value =
serde_json::from_str(&loaded).unwrap();

    // Assert: All fields preserved
    assert_eq!(data["spec_id"], "SPEC-S01-001");
    assert_eq!(data["current_index"], 0);
    assert_eq!(data["quality_gates_enabled"], true);

    // Reconstruct state from loaded data
    let reconstructed =
StateBuilder::new(data["spec_id"].as_str().unwrap())
        .starting_at(SpecStage::Plan)

.quality_gates(data["quality_gates_enabled"].as_bool().unwrap())
        .build();
    assert_eq!(reconstructed.spec_id, state.spec_id);
    assert_eq!(reconstructed.current_index, state.current_index);
}
```

Pattern 2: Pipeline Interrupt → Resume from Checkpoint

Example: S02 - Pipeline Interrupt and Resume

Test (state_persistence_integration_tests.rs:45):

```
#[test]
fn s02_pipeline_interrupt_state_saved_resume_from_checkpoint() {
    let ctx = IntegrationTestContext::new("SPEC-S02-001").unwrap();
    let mut state = StateBuilder::new("SPEC-S02-001")
        .starting_at(SpecStage::Tasks)
        .build();

    // ==================== SAVE CHECKPOINT ====================

    // Act: Save checkpoint before interrupt
    let checkpoint = ctx.commands_dir().join("checkpoint.json");
    std::fs::write(
        &checkpoint,
        json!({
            "spec_id": state.spec_id,
            "checkpoint_index": state.current_index,
            "timestamp": "2025-10-19T10:00:00Z"
        })
        .to_string(),
    )
    .unwrap();

    assert_eq!(state.current_index, 1); // Tasks = index 1

    // ==================== INTERRUPT ====================

    // Simulate interrupt (state dropped)
    drop(state);

    // ==================== RESUME ====================

    // Act: Resume from checkpoint
    let loaded = std::fs::read_to_string(&checkpoint).unwrap();
    let data: serde_json::Value =
serde_json::from_str(&loaded).unwrap();

    let resumed_state = StateBuilder::new("SPEC-S02-001")
        .starting_at(SpecStage::Plan)
        .build();

    // Assert: Checkpoint index preserved
    assert_eq!(data["checkpoint_index"], 1);
    assert_eq!(data["spec_id"], "SPEC-S02-001");

    // Resume would set current_index from checkpoint
    // (not shown: actual resume logic would apply checkpoint)
}
```

Evidence Verification Patterns

Pattern 1: Comprehensive Evidence Verification

```
#[test]
fn verify_complete_stage_evidence() {
    let ctx = IntegrationTestContext::new("SPEC-TEST").unwrap();

    // Simulate complete stage execution
    // ... (write consensus and guardrail artifacts)

    // ==================== VERIFY STRUCTURE ====================

    let verifier = EvidenceVerifier::new(&ctx);

    // Directory structure
    assert!(verifier.assert_structure_valid());

    // ==================== VERIFY CONSENSUS ====================

    // All agents present
    assert!(verifier.assert_consensus_complete(
        SpecStage::Plan,
        &["gemini", "claude", "gpt_pro"]
    ));

    // Individual agents
    assert!(ctx.assert_consensus_exists(SpecStage::Plan, "gemini"));
    assert!(ctx.assert_consensus_exists(SpecStage::Plan, "claude"));
    assert!(ctx.assert_consensus_exists(SpecStage::Plan,
"gpt_pro"));

    // ==================== VERIFY GUARDRAIL ====================

    assert!
(verifier.assert_guardrail_valid(SpecStage::Plan).is_ok());

    // ==================== VERIFY COUNTS ====================

    assert_eq!(ctx.count_consensus_files(), 3);
    assert_eq!(ctx.count_guardrail_files(), 1);
}
```

Pattern 2: Degraded Consensus Detection

```
#[test]
fn verify_degraded_consensus() {
    let ctx = IntegrationTestContext::new("SPEC-TEST").unwrap();

    // Simulate degraded consensus (only 2/3 agents)
    // ... (write only gemini and claude consensus)

    let verifier = EvidenceVerifier::new(&ctx);

    // Should NOT be complete (missing gpt_pro)
    assert!(!verifier.assert_consensus_complete(
        SpecStage::Plan,
        &["gemini", "claude", "gpt_pro"]
```

```rust
    ));

    // But 2/3 is still valid
    assert!(verifier.assert_consensus_complete(
        SpecStage::Plan,
        &["gemini", "claude"]
    ));

    // Verify degraded flag
    let consensus = ctx
        .consensus_dir()
        .join("spec-plan_2025-10-19T10_00_00Z_synthesis.json");
    std::fs::write(
        &consensus,
        json!({"consensus_ok": true, "degraded": true, "missing_agents": ["gpt_pro"]})
            .to_string(),
    )
    .unwrap();

    let content = std::fs::read_to_string(&consensus).unwrap();
    assert!(content.contains("\"degraded\":true"));
}
```

Best Practices

DO

✅ **Use IntegrationTestContext for isolation**:

```rust
#[test]
fn test_workflow() {
    // Each test gets isolated filesystem
    let ctx = IntegrationTestContext::new("SPEC-TEST-001").unwrap();
    // ... test logic
}
```

✅ **Verify evidence at each step**:

```rust
// After consensus
assert!(ctx.assert_consensus_exists(SpecStage::Plan, "gemini"));

// After guardrail
assert!(ctx.assert_guardrail_telemetry_exists(SpecStage::Plan));

// After completion
assert_eq!(ctx.count_consensus_files(), 3);
```

✅ **Test both success and failure paths**:

```rust
#[test]
fn test_success_path() {
    // Happy path
}

#[test]
```

```
fn test_failure_path_with_retry() {
    // Error → Retry → Success
}

#[test]
fn test_failure_path_exhausted_retries() {
    // Error → Retry → Retry → Fail
}
```

✅ **Simulate realistic timing**:

```
let timestamp_attempt1 = "2025-10-19T10:00:00Z";
let timestamp_retry = "2025-10-19T10:05:00Z";  // 5 minutes later

// Evidence shows temporal sequence
```

✅ **Verify state transitions**:

```
assert_eq!(state.current_stage(), Some(SpecStage::Plan));

// Execute stage...

state.current_index += 1;
assert_eq!(state.current_stage(), Some(SpecStage::Tasks));
```

DON'T

❌ **Share IntegrationTestContext across tests**:

```
// Bad: Shared context (tests interfere)
static mut CTX: Option<IntegrationTestContext> = None;

#[test]
fn test_a() {
    unsafe { CTX =
Some(IntegrationTestContext::new("SHARED").unwrap()); }
}

#[test]
fn test_b() {
    unsafe { /* use CTX */ }  // ❌ Flaky (depends on test_a)
}
```

❌ **Test too many stages in one test**:

```
// Bad: Tests entire pipeline (hard to debug failures)
#[test]
fn test_entire_pipeline() {
    // Plan
    // Tasks
    // Implement
    // Validate
    // Audit
    // Unlock
    // → 200 lines, hard to maintain
}
```

```
// Good: Split into focused tests
#[test]
fn w01_plan_stage_workflow() { /* ... */ }

#[test]
fn w02_tasks_stage_workflow() { /* ... */ }
```

✖ **Skip evidence verification**:

```
// Bad: No verification
#[test]
fn test_workflow() {
    // Run workflow...
    // No assertions ✖
}

// Good: Verify evidence
#[test]
fn test_workflow() {
    // Run workflow...
    assert!(ctx.assert_consensus_exists(...));
    assert!(ctx.assert_guardrail_telemetry_exists(...));
}
```

✖ **Use hard-coded paths**:

```
// Bad: Hard-coded paths (breaks on other machines)
let consensus = Path::new("/tmp/consensus/SPEC-TEST/plan.json");

// Good: Use IntegrationTestContext
let consensus = ctx.consensus_dir().join("plan.json");
```

Running Integration Tests

Run All Integration Tests

```
cd codex-rs
cargo test --test '*_integration_tests'
```

Runs: - `workflow_integration_tests.rs` - `error_recovery_integration_tests.rs` - `state_persistence_integration_tests.rs` - `quality_flow_integration_tests.rs` - `concurrent_operations_integration_tests.rs`

Run Specific Category

```
# Workflow tests only
cargo test --test workflow_integration_tests

# Error recovery tests only
cargo test --test error_recovery_integration_tests
```

Run Specific Test

```
cargo test --test workflow_integration_tests w01_plan_stage_complete_workflow
```

Run with Output

```
cargo test --test workflow_integration_tests -- --nocapture
```

Shows `println!()` output for debugging.

Summary

Integration Testing Best Practices:

1. **Isolation**: Use `IntegrationTestContext` for each test
2. **Evidence**: Verify evidence at each step
3. **Coverage**: Test success and failure paths
4. **Clarity**: One workflow per test
5. **Timing**: Simulate realistic sequences
6. **State**: Verify state transitions
7. **Cleanup**: Automatic (TempDir drops)

Test Categories: - ✅ W01-W15: Workflow integration (stage workflows, pipelines) - ✅ E01-E15: Error recovery (retry, fallback, degradation) - ✅ S01-S10: State persistence (serialize, resume, checkpoint) - ✅ Q01-Q10: Quality gates (BeforeSpecify, AfterSpecify, AfterTasks) - ✅ C01-C10: Concurrent operations (parallel, locking)

Key Patterns: - ✅ Multi-module workflows (Handler → Consensus → Evidence → Guardrail → State) - ✅ Error propagation (Failure → Retry → Recovery → Evidence) - ✅ State persistence (Serialize → Load → Reconstruct) - ✅ Evidence verification (EvidenceVerifier, counts, structure)

Next Steps: - E2E Testing Guide - Complete user workflows - Property Testing Guide - Generative invariant testing - Test Infrastructure - MockMcpManager, fixtures

---

References: - Workflow tests: `codex-rs/tui/tests/workflow_integration_tests.rs` - Error recovery: `codex-rs/tui/tests/error_recovery_integration_tests.rs` - State persistence: `codex-rs/tui/tests/state_persistence_integration_tests.rs` - IntegrationTestContext: `codex-rs/tui/tests/common/integration_harness.rs`

Performance Testing Guide

Comprehensive guide to performance testing, benchmarking, and profiling.

Overview

Performance Testing Philosophy: Measure, don’t guess. Validate optimizations with data.

Goals: - Measure baseline performance - Validate optimizations - Detect regressions - Identify bottlenecks

Tools: - **criterion**: Statistical benchmarking - **cargo-flamegraph**: Profiling - **cargo-bloat**: Binary size analysis - **hyperfine**: Command-line benchmarking

Current Benchmarks: - Database performance (6.6× read speedup validated) - MCP client (5.3× faster than subprocess validated) - Connection pooling (R2D2)

Benchmarking with Criterion

What is Criterion?

Criterion is a statistical benchmarking tool for Rust that provides: - Accurate measurements (micro/nanosecond precision) - Statistical analysis (mean, stddev, outliers) - Regression detection (compare to baseline) - HTML reports with charts

Website: https://bheisler.github.io/criterion.rs/

Setup

Add to Cargo.toml:

```toml
[dev-dependencies]
criterion = { version = "0.5", features = ["html_reports"] }

[[bench]]
name = "my_benchmark"
harness = false
```

Basic Benchmark

File: `benches/simple_benchmark.rs`

```rust
use criterion::{Criterion, black_box, criterion_group,
criterion_main};

fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 1,
        1 => 1,
        n => fibonacci(n - 1) + fibonacci(n - 2),
    }
}

fn benchmark_fibonacci(c: &mut Criterion) {
    c.bench_function("fib 20", |b| {
```

```
                b.iter(|| fibonacci(black_box(20)));
            });
        }

        criterion_group!(benches, benchmark_fibonacci);
        criterion_main!(benches);
```

Run:

```
        cargo bench --bench simple_benchmark
```

Output:

```
fib 20                  time:   [26.029 µs 26.251 µs 26.509 µs]
Found 11 outliers among 100 measurements (11.00%)
  6 (6.00%) high mild
  5 (5.00%) high severe
```

Database Performance Benchmark

Example: `codex-rs/core/benches/db_performance.rs`

Performance Targets: - Before: 850µs/read, 2.1ms/write, 78ms/100-read batch - After: 129µs/read, 0.9ms/write, 12ms/100-read batch - Improvement: 6.6× read, 2.3× write, 6.5× batch

---

Benchmark Setup

```
        use criterion::{Criterion, Throughput, black_box, criterion_group, criterion_main};
        use codex_core::db::initialize_pool;
        use tempfile::TempDir;

        /// Create temporary database with schema
        fn setup_temp_db() -> (TempDir, PathBuf) {
            let temp_dir = TempDir::new().expect("Failed to create temp dir");
            let db_path = temp_dir.path().join("test.db");

            let conn = Connection::open(&db_path).expect("Failed to open connection");
            conn.execute_batch(
                "CREATE TABLE IF NOT EXISTS consensus_runs (
                    id INTEGER PRIMARY KEY,
                    spec_id TEXT NOT NULL,
                    stage TEXT NOT NULL,
                    consensus_ok INTEGER NOT NULL,
                    created_at INTEGER NOT NULL
                );
                CREATE INDEX IF NOT EXISTS idx_spec_stage ON consensus_runs(spec_id, stage);"
            )
            .expect("Failed to create schema");

            (temp_dir, db_path)
        }

        /// Create connection pool with WAL mode
        fn setup_pool(db_path: &PathBuf) -> Pool<SqliteConnectionManager> {
            initialize_pool(db_path, 10).expect("Failed to initialize pool")
        }
```

Benchmark #1: Connection Pool vs Single Connection

```
        fn benchmark_connection_pool_vs_single(c: &mut Criterion) {
            let mut group = c.benchmark_group("connection_pool_vs_single");

            // Setup: Create database with test data
            let (_temp_dir, db_path) = setup_temp_db();
            let pool = setup_pool(&db_path);

            // Insert 1000 test records
            {
                let conn = pool.get().expect("Failed to get connection");
                insert_test_data(&conn, 1000);
            }

            // Benchmark: Pooled connection reads
            group.bench_function("pooled_connection_read", |b| {
                b.iter(|| {
                    let conn = pool.get().expect("Failed to get
connection");
                    let mut stmt = conn
                        .prepare("SELECT * FROM consensus_runs WHERE spec_id
= ?1")
                        .expect("Failed to prepare statement");
                    let _count = stmt
                        .query_map(["SPEC-TEST-050"], |_row| Ok(()))
                        .expect("Failed to query")
                        .count();
                    black_box(_count);
                });
            });

            // Benchmark: Single connection reads (reused connection)
            group.bench_function("single_connection_read", |b| {
                let conn = setup_single_connection_wal(&db_path);
                b.iter(|| {
                    let mut stmt = conn
                        .prepare("SELECT * FROM consensus_runs WHERE spec_id
= ?1")
                        .expect("Failed to prepare statement");
                    let _count = stmt
                        .query_map(["SPEC-TEST-050"], |_row| Ok(()))
                        .expect("Failed to query")
                        .count();
                    black_box(_count);
                });
            });

            group.finish();
        }
```

Results:

```
connection_pool_vs_single/pooled_connection_read
                        time:   [129.45 µs 130.12 µs 130.89 µs]
```

```
connection_pool_vs_single/single_connection_read
                        time:   [127.89 µs 128.45 µs 129.12 µs]
```

Analysis: - ✅ Pool overhead minimal (~1-2µs) - ✅ Both achieve target (<150µs vs 850µs before) - ✅ 6.6× improvement validated

---

Benchmark #2: WAL Mode Impact

```
        fn benchmark_wal_mode_impact(c: &mut Criterion) {
            let mut group = c.benchmark_group("wal_mode_impact");

            let (_temp_dir, db_path) = setup_temp_db();

            // Setup: Connection with WAL mode
            let conn_wal = setup_single_connection_wal(&db_path);
            insert_test_data(&conn_wal, 1000);

            // Setup: Connection with DELETE mode (no WAL)
            let (_temp_dir2, db_path2) = setup_temp_db();
            let conn_delete = setup_single_connection_delete(&db_path2);
            insert_test_data(&conn_delete, 1000);

            // Benchmark: Read with WAL
            group.bench_function("read_wal", |b| {
                b.iter(|| {
                    let mut stmt = conn_wal
                        .prepare("SELECT * FROM consensus_runs WHERE spec_id
= ?1")
                        .unwrap();
                    black_box(stmt.query_map(["SPEC-TEST-050"], |_|
Ok(())).unwrap().count());
                });
            });

            // Benchmark: Read with DELETE mode
            group.bench_function("read_delete", |b| {
                b.iter(|| {
                    let mut stmt = conn_delete
                        .prepare("SELECT * FROM consensus_runs WHERE spec_id
= ?1")
                        .unwrap();
                    black_box(stmt.query_map(["SPEC-TEST-050"], |_|
Ok(())).unwrap().count());
                });
            });

            group.finish();
        }
```

Results:

```
wal_mode_impact/read_wal
                        time:   [129.12 µs 130.45 µs 131.89 µs]

wal_mode_impact/read_delete
                        time:   [847.34 µs 851.23 µs 856.78 µs]

Improvement: 6.58× faster with WAL ✅
```

Throughput Benchmarks

Pattern: Measure operations per second

```
fn benchmark_batch_reads(c: &mut Criterion) {
    let mut group = c.benchmark_group("batch_reads");

    let (_temp_dir, db_path) = setup_temp_db();
    let pool = setup_pool(&db_path);
    let conn = pool.get().unwrap();
    insert_test_data(&conn, 1000);

    // Benchmark 100 reads (measure throughput)
    group.throughput(Throughput::Elements(100));
    group.bench_function("read_100", |b| {
        b.iter(|| {
            for i in 0..100 {
                let conn = pool.get().unwrap();
                let mut stmt = conn.prepare("SELECT * FROM consensus_runs WHERE spec_id = ?1").unwrap();
                let _count = stmt.query_map([format!("SPEC-TEST-{:03}", i % 100)], |_| Ok(())).unwrap().count();
                black_box(_count);
            }
        });
    });

    group.finish();
}
```

Results:

```
batch_reads/read_100    time:   [12.234 ms 12.456 ms 12.689 ms]
                        thrpt:  [7.88 Kelem/s 8.03 Kelem/s 8.17 Kelem/s]

Before optimization: 78ms/100 reads → 1.28 Kelem/s
After optimization:  12ms/100 reads → 8.03 Kelem/s
Improvement: 6.27× faster ✅
```

Running Benchmarks

Run all benchmarks:

```
cd codex-rs
cargo bench
```

Run specific benchmark:

```
cargo bench --bench db_performance
```

Run specific function:

```
cargo bench --bench db_performance -- connection_pool
```

Generate baseline (for regression detection):

```
cargo bench -- --save-baseline baseline_2025_11_17
```

Compare to baseline:

```
cargo bench -- --baseline baseline_2025_11_17
```

View HTML reports:

```
open target/criterion/report/index.html
```

Profiling

Flamegraphs with cargo-flamegraph

What are Flamegraphs?: - Visual representation of stack traces - Shows where CPU time is spent - Width = time spent in function - Height = call stack depth

Install:

```
cargo install flamegraph
```

Usage:

```
# Profile specific benchmark
cargo flamegraph --bench db_performance -- --bench

# Profile specific test
cargo flamegraph --test integration_test

# Profile binary
cargo flamegraph --bin code
```

Output: `flamegraph.svg` (interactive SVG)

Interpretation: - **Wide bars**: Hot paths (optimize these) - **Narrow bars**: Not worth optimizing - **Tall stacks**: Deep call chains

---

perf (Linux only)

Install:

```
sudo apt install linux-tools-generic
```

Record:

```
cargo build --release
perf record --call-graph=dwarf ./target/release/code
```

Analyze:

```
perf report
```

Generate Flamegraph:

```
perf script | stackcollapse-perf.pl | flamegraph.pl > perf.svg
```

---

cargo-bloat (Binary Size Analysis)

Purpose: Find large dependencies

Install:

```
cargo install cargo-bloat
```

Usage:

```
cd codex-rs
cargo bloat --release
```

Output:

```
File  .text     Size Crate
0.7%   1.2%   24.5KiB regex
0.6%   1.0%   20.1KiB serde_json
0.5%   0.9%   18.7KiB tokio
...
```

Optimize (if needed):

```
# Cargo.toml
[profile.release]
lto = true              # Link-time optimization
codegen-units = 1       # Better optimization, slower build
strip = true            # Strip symbols
opt-level = "z"         # Optimize for size
```

Command-Line Benchmarking

hyperfine

Purpose: Benchmark CLI commands

Install:

```
cargo install hyperfine
```

Usage:

```
# Benchmark single command
hyperfine './codex-rs/target/release/code --version'

# Compare commands
hyperfine \
  './codex-rs/target/release/code doctor' \
  './codex-rs/target/dev-fast/code doctor'

# Warmup runs
hyperfine --warmup 3 'cargo test'

# Multiple runs
hyperfine --runs 100 './codex-rs/target/release/code --help'
```

Example Output:

```
Benchmark 1: ./target/release/code --version
  Time (mean ± σ):      12.3 ms ±   0.5 ms    [User: 8.2 ms, System:
3.1 ms]
  Range (min … max):    11.5 ms …  14.2 ms     100 runs
```

Benchmarking /speckit.auto

Example:

```
hyperfine --warmup 1 --runs 5 \
  './codex-rs/target/release/code run "/speckit.auto SPEC-TEST-001"'
```

Expected:

```
Time (mean ± σ):     45.2 s ±  2.1 s    [User: 38.1 s, System: 3.2
s]
Range (min … max):   42.8 s … 48.5 s    5 runs
```

Performance Metrics

Database Performance

Measured Metrics: - Read latency (µs): 850 → 129 (6.6× improvement) - Write latency (ms): 2.1 → 0.9 (2.3× improvement) - Batch reads (ms/100): 78 → 12 (6.5× improvement)

How Measured:

```
// codex-rs/core/benches/db_performance.rs
criterion_group!(benches,
    benchmark_connection_pool_vs_single,
    benchmark_wal_mode_impact,
    benchmark_batch_reads,
);
```

MCP Performance

Measured Metrics: - Native MCP client: 8.7ms typical - Subprocess MCP: 46ms typical - Improvement: 5.3× faster

How Measured:

```
// Integration test timing
let start = std::time::Instant::now();
let result = mcp_client.call_tool(...).await?;
let elapsed = start.elapsed();
assert!(elapsed < Duration::from_millis(10)); // <10ms
```

Config Hot-Reload

Measured Metrics: - Reload latency (p95): <100ms - File watch overhead: <1% CPU

How Measured:

```
// Integration test
let start = std::time::Instant::now();
// Modify config file
std::fs::write(&config_path, new_content)?;
// Wait for reload
```

```
tokio::time::sleep(Duration::from_millis(50)).await;
// Verify reload
assert_eq!(app.current_model(), "gpt-5-medium");
let elapsed = start.elapsed();
assert!(elapsed < Duration::from_millis(100));
```

Regression Testing

Baseline Comparison

Save baseline:

```
cargo bench -- --save-baseline v1.0.0
```

Compare:

```
# After changes
cargo bench -- --baseline v1.0.0
```

Output:

```
connection_pool_vs_single/pooled_connection_read
                        time:   [129.45 µs 130.12 µs 130.89 µs]
                        change: [-0.5% +0.2% +1.1%] (p = 0.23 >
0.05)
                        No change in performance detected.
```

Interpretation: - Change <5%: No regression - Change >5%: Investigate - Change >10%: **Regression detected** (fix before merge)

Continuous Performance Monitoring

CI Integration (future):

```
# .github/workflows/performance.yml
jobs:
  benchmark:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Run benchmarks
        run: cargo bench -- --save-baseline ci-baseline

      - name: Compare to previous
        run: cargo bench -- --baseline ci-baseline-previous

      - name: Fail if regression >10%
        run: |
          if grep "change:.*[+][1-9][0-9]"
target/criterion/**/new/estimates.txt; then
            echo "Performance regression detected!"
            exit 1
          fi
```

Best Practices

DO

✅ **Measure before optimizing**:

```bash
# Before: Measure baseline
cargo bench -- --save-baseline before_optimization

# Optimize code...

# After: Measure improvement
cargo bench -- --baseline before_optimization
```

✅ **Use `black_box()` to prevent optimization**:

```rust
// Good: Prevents compiler from optimizing away
b.iter(|| {
    black_box(expensive_function(black_box(input)));
});

// Bad: Compiler might optimize this away
b.iter(|| {
    expensive_function(input);
});
```

✅ **Benchmark realistic workloads**:

```rust
// Good: Real-world data
let data = load_fixture("real_prd.md");
b.iter(|| detect_ambiguities(black_box(&data)));

// Bad: Trivial input
let data = "test";
b.iter(|| detect_ambiguities(black_box(&data)));
```

✅ **Run benchmarks on consistent hardware**: - Same machine (or CI runner) - Close other programs - Disable CPU frequency scaling (if possible)

✅ **Set performance targets**:

```rust
// Document targets in benchmark comments
/// Target: <150μs (was 850μs before optimization)
group.bench_function("pooled_read", |b| { ... });
```

DON’T

❌ **Optimize without measuring**:

```rust
// Bad: Premature optimization
// "This looks slow, let me rewrite it"

// Good: Measure first
// cargo bench → identify hot path → optimize
```

❌ Trust microbenchmarks for macro performance:

```
// Bad: Optimizing single function
fn fast_function() { /* 1μs faster */ }

// Better: Benchmark complete workflow
fn complete_pipeline() { /* Does 1μs matter here? */ }
```

❌ Ignore variance:

```
# Bad: "It ran in 10ms once"

# Good: "Mean: 10.2ms ± 0.3ms (100 runs)"
```

❌ Benchmark in debug mode:

```
# Bad: Debug mode (100× slower)
cargo bench

# Good: Release mode (default for benches)
cargo bench --release
```

Summary

Performance Testing Best Practices:

1. **Measure**: Use criterion for accurate benchmarks
2. **Profile**: Use flamegraphs to find hot paths
3. **Validate**: Confirm optimizations with data
4. **Regress**: Detect performance regressions
5. **Target**: Set clear performance goals

Tools: - ✅ criterion (statistical benchmarking) - ✅ cargo-flamegraph (profiling) - ✅ cargo-bloat (binary size) - ✅ hyperfine (CLI benchmarking) - ✅ perf (Linux profiling)

Validated Improvements: - ✅ Database: 6.6× read, 2.3× write - ✅ MCP: 5.3× faster (8.7ms vs 46ms) - ✅ Config reload: <100ms (p95)

Key Metrics: - ✅ Latency (μs, ms, s) - ✅ Throughput (ops/sec, elem/sec) - ✅ Percentiles (p50, p95, p99) - ✅ Variance (stddev, outliers)

Next Steps: - Testing Strategy - Overall testing approach - CI/CD Integration - Automated testing - Test Infrastructure - MockMcpManager, fixtures

---

References: - criterion: https://bheisler.github.io/criterion.rs/ - Database benchmarks: `codex-rs/core/benches/db_performance.rs` - Profiling guide: https://nnethercote.github.io/perf-book/ - hyperfine: https://github.com/sharkdp/hyperfine

---

Property-Based Testing Guide

Comprehensive guide to property-based testing with proptest.

Overview

Property-Based Testing Philosophy: Generate random inputs to verify invariants hold across all possible values

Tool: proptest (Rust equivalent of QuickCheck/Hypothesis)

Goals: - Test invariants (properties that always hold) - Find edge cases automatically - Verify mathematical properties - Reduce test boilerplate

Current Status: - ~30 property-based tests - 100% pass rate - 100 test cases per property (default) - Integrated with standard test suite

---

What is Property-Based Testing?

Traditional Example-Based Testing

```
#[test]
fn test_reverse_twice_is_identity() {
    let vec = vec![1, 2, 3];
    let reversed = reverse(reverse(vec.clone()));
    assert_eq!(reversed, vec);
}
```

Limitations: - Only tests one input (`[1, 2, 3]`) - May miss edge cases (empty, single element, duplicates) - Requires manual case selection

---

Property-Based Testing

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_reverse_twice_is_identity(vec in any::<Vec<i32>>()) {
        let reversed = reverse(reverse(vec.clone()));
        prop_assert_eq!(reversed, vec);
    }
}
```

Benefits: - ✅ Tests 100 random inputs automatically - ✅ Finds edge cases (empty, single, large, etc.) - ✅ Shrinks failing input to minimal case - ✅ Focuses on **properties** not **examples**

Getting Started

Add proptest Dependency

Cargo.toml:

```
[dev-dependencies]
proptest = "1.3"
```

Basic Property Test

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_addition_commutative(a in any::<i32>(), b in any::<i32>
()) {
        // Property: a + b == b + a
        prop_assert_eq!(a + b, b + a);
    }
}
```

How it works: 1. Generate 100 random pairs of `(a, b)` 2. Run test with each pair 3. If any fails, shrink to minimal failing case 4. Report failure with minimal input

---

Generators

Built-in Generators

Primitive Types:

```
proptest! {
    #[test]
    fn test_primitives(
        n in any::<i32>(),
        s in any::<String>(),
        b in any::<bool>(),
    ) {
        // Test with random primitives
    }
}
```

Collections:

```
proptest! {
    #[test]
    fn test_collections(
        vec in any::<Vec<i32>>(),
        set in any::<HashSet<String>>(),
        map in any::<HashMap<i32, String>>(),
    ) {
        // Test with random collections
    }
}
```

Ranges:

```
proptest! {
    #[test]
    fn test_ranges(
        index in 0usize..10,          // 0-9
        score in 0.0..100.0,          // 0.0-99.999...
        percentage in 0..=100,        // 0-100 (inclusive)
    ) {
        prop_assert!(index < 10);
        prop_assert!(score < 100.0);
        prop_assert!(percentage <= 100);
    }
}
```

Custom Generators

Regex Patterns:

```
proptest! {
    #[test]
    fn test_spec_id_format(
        spec_id in "[A-Z]{4}-[A-Z]{3}-[0-9]{3}"
    ) {
        // Generates: "SPEC-KIT-001", "ABCD-XYZ-999", etc.
        prop_assert!(is_valid_spec_id(&spec_id));
    }
}
```

Custom Strategies:

```
fn spec_stage_strategy() -> impl Strategy<Value = SpecStage> {
    prop_oneof![
        Just(SpecStage::Plan),
        Just(SpecStage::Tasks),
        Just(SpecStage::Implement),
        Just(SpecStage::Validate),
        Just(SpecStage::Audit),
        Just(SpecStage::Unlock),
    ]
}

proptest! {
    #[test]
    fn test_stage_valid(stage in spec_stage_strategy()) {
        // Tests all 6 stages
        prop_assert!(is_valid_stage(&stage));
    }
}
```

Testing Invariants

Invariant 1: State Index Always Valid

Property: State index ∈ [0, 5] → `current_stage()` returns `Some(_)`, else `None`

Test (property_based_tests.rs:21):

```
proptest! {
    #[test]
    fn pb01_state_index_always_in_valid_range(index in 0usize..20) {
        let mut state = StateBuilder::new("SPEC-PB01-TEST")
            .starting_at(SpecStage::Plan)
            .build();

        state.current_index = index;

        // Invariant: index ∈ [0, 5] → Some(_), else None
        if index < 6 {
            prop_assert!(state.current_stage().is_some());
        } else {
            prop_assert_eq!(state.current_stage(), None);
        }
    }
}
```

What This Tests: - ✅ All indices 0-19 handled correctly - ✅ Valid indices (0-5) return Some - ✅ Invalid indices (6+) return None - ✅ No panics or crashes

---

Invariant 2: Current Stage Mapping

Property: For index ∈ [0, 5], `current_stage()` returns correct stage

Test (property_based_tests.rs:38):

```
proptest! {
    #[test]
    fn pb02_current_stage_always_some_when_index_under_six(
        index in 0usize..6
    ) {
        let mut state = StateBuilder::new("SPEC-PB02-TEST").build();
        state.current_index = index;

        prop_assert!(state.current_stage().is_some());

        // Verify correct stage mapping
        let expected_stages = vec![
            SpecStage::Plan,
            SpecStage::Tasks,
            SpecStage::Implement,
            SpecStage::Validate,
            SpecStage::Audit,
            SpecStage::Unlock,
        ];

        prop_assert_eq!(
            state.current_stage(),
            Some(expected_stages[index])
        );
    }
}
```

What This Tests: - ✅ All valid indices (0-5) return Some - ✅ Correct stage for each index - ✅ Consistent mapping

---

Invariant 3: Retry Count Never Negative

Property: Retry count ≤ max_retries (capped at max)

Test (property_based_tests.rs:62):

```
        proptest! {
            #[test]
            fn pb03_retry_count_never_negative(retries in 0usize..100) {
                let ctx = IntegrationTestContext::new("SPEC-PB03-
TEST").unwrap();

                let max_retries = 3;
                let capped_retries = retries.min(max_retries);

                let retry_file = ctx.commands_dir().join("retry.json");
                std::fs::write(&retry_file, json!({
                    "retry_count": capped_retries,
                    "max_retries": max_retries,
                    "within_limit": capped_retries <= max_retries
                }).to_string()).unwrap();

                let content = std::fs::read_to_string(&retry_file).unwrap();
                let data: serde_json::Value =
serde_json::from_str(&content).unwrap();

                prop_assert!(data["retry_count"].as_u64().unwrap() <=
max_retries as u64);
                prop_assert_eq!(data["within_limit"].as_bool(), Some(true));
            }
        }
```

What This Tests: - ✅ Retry counts 0-99 all capped correctly - ✅ No retry count exceeds max - ✅ within_limit flag always true

---

Testing Evidence Integrity

Property 1: Written Evidence Always Parseable JSON

Property: Any evidence written is valid JSON

Test (property_based_tests.rs:90):

```
        proptest! {
            #[test]
            fn pb04_written_evidence_always_parseable_json(
                agent in "[a-z]{3,10}",
                content in ".*"
            ) {
                let ctx = IntegrationTestContext::new("SPEC-PB04-
TEST").unwrap();

                let evidence = json!({
                    "agent": agent,
                    "content": content,
                    "timestamp": "2025-10-19T00:00:00Z"
                });

                let file = ctx.consensus_dir().join("test.json");
                std::fs::write(&file, evidence.to_string()).unwrap();
                // Invariant: File is valid JSON
                let content = std::fs::read_to_string(&file).unwrap();
                let parsed: Value = serde_json::from_str(&content).unwrap();

                prop_assert_eq!(parsed["agent"].as_str(),
Some(agent.as_str()));
            }
        }
```

What This Tests: - ✅ Random agent names (3-10 lowercase letters) - ✅ Random content (any string) - ✅ Always produces valid JSON - ✅ Round-trip serialization works

---

Property 2: Evidence File Names Valid

Property: Generated filenames are valid filesystem paths

```
        proptest! {
            #[test]
            fn pb05_evidence_filenames_always_valid(
                spec_id in "[A-Z]{4}-[A-Z]{3}-[0-9]{3}",
                stage in spec_stage_strategy(),
                agent in "[a-z]{5,10}",
            ) {
                let filename = format!(
                    "spec-{:?}_{}_{}_{}.json",
                    stage,
                    spec_id,
                    "2025-10-19T10_00_00Z",
                    agent
                );

                // Invariant: Filename contains no invalid characters
                prop_assert!(!filename.contains('/'));
                prop_assert!(!filename.contains('\\'));
                prop_assert!(!filename.contains('\0'));

                // Invariant: Filename is not empty
                prop_assert!(!filename.is_empty());

                // Invariant: Filename has .json extension
                prop_assert!(filename.ends_with(".json"));
            }
        }
```

What This Tests: - ✅ Random SPEC IDs - ✅ All 6 stages - ✅ Random agent names - ✅ Filenames always valid (no /, \, null bytes) - ✅ Always has .json extension

---

Testing Collections

Property 1: Filtering Never Increases Length

Property: Filtered collection ≤ original length

```
        proptest! {
```

```
    #[test]
    fn test_filter_never_increases_length(
        vec in any::<Vec<i32>>()
    ) {
        let filtered: Vec<_> = vec.iter()
            .filter(|&&x| x > 0)
            .collect();

        prop_assert!(filtered.len() <= vec.len());
    }
}
```

Property 2: Sorting Preserves Length

Property: Sorted collection has same length as original

```
proptest! {
    #[test]
    fn test_sort_preserves_length(
        mut vec in any::<Vec<i32>>()
    ) {
        let original_len = vec.len();

        vec.sort();

        prop_assert_eq!(vec.len(), original_len);
    }
}
```

Property 3: Dedupe Length

Property: Deduplicated length ≤ original length

```
proptest! {
    #[test]
    fn test_dedupe_length(
        mut vec in any::<Vec<i32>>()
    ) {
        let original_len = vec.len();

        vec.sort();
        vec.dedup();

        prop_assert!(vec.len() <= original_len);
    }
}
```

Testing String Operations

Property 1: Truncation Length

Property: Truncated string ≤ max length (plus ellipsis)

```
proptest! {
    #[test]
    fn test_truncate_length(
        text in any::<String>(),
        max_len in 1usize..100,
    ) {
        let truncated = truncate_context(&text, max_len);

        if text.len() <= max_len {
            // No truncation
            prop_assert_eq!(truncated.len(), text.len());
        } else {
            // Truncated with "..."
            prop_assert_eq!(truncated.len(), max_len + 3);
        }
    }
}
```

Property 2: Regex Escape Safety

Property: Escaped string never causes regex parse error

```
proptest! {
    #[test]
    fn test_regex_escape_never_panics(s in ".*") {
        let escaped = regex_escape(&s);

        // Invariant: Escaped string is valid regex literal
        let pattern = format!("^{}$", escaped);
        let re = Regex::new(&pattern);

        prop_assert!(re.is_ok());
    }
}
```

Shrinking

What is Shrinking?

When a property test fails, proptest **shrinks** the failing input to the **minimal** failing case.

Example:

```
proptest! {
    #[test]
    fn test_all_positive(vec in any::<Vec<i32>>()) {
        prop_assert!(vec.iter().all(|&x| x > 0));
    }
}
```

Failure:

```
Test failed for input: [1, 2, 3, 0, 5, 6, 7, 8, 9]
Shrinking...
Minimal failing input: [0]
```

Shrinking Example

Original failure: - Input: `vec = [42, -17, 0, 99, -3, 100, 256, -1, 7]` - Failed because: -17, -3, -1 are negative

After shrinking: - Input: `vec = [-1]` - Still fails, but minimal

Benefits: - ✅ Easier to debug - ✅ Clear failure reason - ✅ No noise from extra elements

---

Advanced Patterns

Conditional Properties

Pattern: Property holds only under certain conditions

```
proptest! {
    #[test]
    fn test_division_inverse(
        a in any::<f64>(),
        b in any::<f64>()
    ) {
        // Property only holds when b ≠ 0
        prop_assume!(b != 0.0);

        let result = a / b * b;
        prop_assert!((result - a).abs() < 0.0001);
    }
}
```

`prop_assume!(condition)`: - Skips test case if condition false - Generates new random input - Useful for preconditions

---

Composite Strategies

Pattern: Combine multiple generators

```
fn state_and_index_strategy() -> impl Strategy<Value = (SpecAutoState, usize)> {
    (spec_id_strategy(), 0usize..20)
        .prop_map(|(spec_id, index)| {
            let mut state = StateBuilder::new(&spec_id).build();
            state.current_index = index;
            (state, index)
        })
}

proptest! {
    #[test]
    fn test_with_composite(
        (state, index) in state_and_index_strategy()
    ) {
        if index < 6 {
            prop_assert!(state.current_stage().is_some());
        }
    }
}
```

Regression Testing

Pattern: Save failing inputs, re-test on every run

File: `proptest-regressions/property_based_tests.txt`

```
# Seeds for failure cases
xs 1234567890
xs 9876543210
```

Usage: 1. Test fails with input `xs = 1234567890` 2. proptest saves seed to regression file 3. Next run always tests that seed first 4. Ensures bug doesn't resurface

---

Configuration

Adjust Test Cases

Default: 100 test cases per property

Custom:

```
proptest! {
    #![proptest_config(ProptestConfig::with_cases(1000))]

    #[test]
    fn test_with_more_cases(n in any::<i32>()) {
        // Runs 1000 times instead of 100
    }
}
```

Environment Variable

```
# Run 10,000 test cases
PROPTEST_CASES=10000 cargo test --test property_based_tests
```

Timeout

```
proptest! {
    #![proptest_config(ProptestConfig {
        cases: 100,
        max_shrink_iters: 10000,
        timeout: 5000,  // 5 seconds
        .. ProptestConfig::default()
    })]

    #[test]
    fn test_with_timeout(vec in any::<Vec<i32>>()) {
        // Timeout if takes >5s
    }
}
```

Best Practices

DO

✅ **Test invariants, not examples**:

```
// Good: Tests property
proptest! {
    #[test]
    fn test_reverse_twice_identity(vec in any::<Vec<i32>>()) {
        prop_assert_eq!(reverse(reverse(vec.clone())), vec);
    }
}

// Bad: Tests specific example (use regular #[test])
proptest! {
    #[test]
    fn test_specific_case() {
        let vec = vec![1, 2, 3];
        prop_assert_eq!(reverse(reverse(vec.clone())), vec);
    }
}
```

✅ **Use `prop_assume!()` for preconditions**:

```
proptest! {
    #[test]
    fn test_with_precondition(
        index in 0usize..100,
        vec in any::<Vec<i32>>()
    ) {
        prop_assume!(index < vec.len());

        let elem = vec[index];
        // Test with valid index
    }
}
```

✅ **Test mathematical properties**:

```
proptest! {
    #[test]
    fn test_addition_associative(a in any::<i32>(), b in any::<i32>
(), c in any::<i32>()) {
        prop_assert_eq!((a + b) + c, a + (b + c));
    }

    #[test]
    fn test_multiplication_distributive(a in any::<i32>(), b in
any::<i32>(), c in any::<i32>()) {
        prop_assert_eq!(a * (b + c), a * b + a * c);
    }
}
```

✅ **Test round-trip properties**:

```
proptest! {
```

```
            #[test]
            fn test_serialize_deserialize(state in any::<SpecAutoState>()) {
                let json = serde_json::to_string(&state).unwrap();
                let deserialized: SpecAutoState =
serde_json::from_str(&json).unwrap();

                prop_assert_eq!(deserialized, state);
            }
        }
```

DON'T

✗ Test concrete outputs:

```
        // Bad: Property tests shouldn't check specific outputs
        proptest! {
            #[test]
            fn test_bad(n in any::<i32>()) {
                prop_assert_eq!(add_one(n), n + 1);  // ✗ This is just
example-based
            }
        }
```

✗ Generate invalid inputs:

```
        // Bad: Generates many invalid cases (slow)
        proptest! {
            #[test]
            fn test_with_many_assumes(
                a in any::<i32>(),
                b in any::<i32>(),
            ) {
                prop_assume!(a > 0);
                prop_assume!(b > 0);
                prop_assume!(a < b);
                prop_assume!(b % 2 == 0);
                // ... many assumes = slow
            }
        }

        // Good: Use constrained generator
        fn even_positive_pair_strategy() -> impl Strategy<Value = (i32,
i32)> {
            (1i32..1000, 1i32..1000)
                .prop_filter("a < b and b even", |(a, b)| a < b && b % 2 ==
0)
        }
```

Running Property Tests

Run All Property Tests

```
        cd codex-rs
        cargo test --test property_based_tests
```

Run with More Cases

```
PROPTEST_CASES=1000 cargo test --test property_based_tests
```

Debug Failing Test

```
# Run specific property test
cargo test --test property_based_tests pb01_state_index

# With verbose output
cargo test --test property_based_tests pb01_state_index -- --nocapture
```

Re-run Regression Cases

```
# Automatically runs saved regression cases from proptest-regressions/
cargo test --test property_based_tests
```

Summary

Property-Based Testing Best Practices:

1. **Invariants**: Test properties that always hold
2. **Generators**: Use appropriate generators (ranges, regex, custom)
3. **Shrinking**: Let proptest find minimal failing case
4. **Preconditions**: Use `prop_assume!()` for preconditions
5. **Configuration**: Adjust test cases with `PROPTEST_CASES`
6. **Regression**: Save failing cases automatically

Common Properties to Test: - ✅ Invariants (index bounds, retry limits) - ✅ Round-trip (serialize → deserialize) - ✅ Mathematical (associativity, commutativity, distributivity) - ✅ Collection operations (filter length, sort preserves length) - ✅ String operations (truncate length, regex escape safety) - ✅ Evidence integrity (valid JSON, valid filenames)

Key Concepts: - ✅ Generators create random inputs - ✅ Shrinking finds minimal failing case - ✅ Regression tests prevent regressions - ✅ 100 test cases per property (default)

Next Steps: - CI/CD Integration - Automated testing pipeline - Performance Testing - Benchmarks and profiling - Test Infrastructure - MockMcpManager, fixtures

---

References: - proptest docs: https://docs.rs/proptest - Property tests: `codex-rs/tui/tests/property_based_tests.rs` - Regression files: `proptest-regressions/`

Test Infrastructure

Comprehensive testing infrastructure for the codebase.

Overview

Test Infrastructure Components: - **MockMcpManager**: Mock MCP server for isolated testing - **IntegrationTestContext**: Multi-module test harness - **StateBuilder**: Test state configuration - **EvidenceVerifier**: Artifact validation helpers - **Fixture Library**: Real production data (20 files, 96 KB) - **Coverage Tools**: cargo-tarpaulin, cargo-llvm-cov - **Property Testing**: proptest for generative testing

Location: `codex-rs/tui/tests/common/` (shared test utilities)

Purpose: Enable comprehensive testing without external dependencies

---

MockMcpManager

Purpose

Mock implementation of `McpConnectionManager` for testing MCP-dependent code without requiring a live local-memory server.

Location: `codex-rs/tui/tests/common/mock_mcp.rs` (272 LOC)

Use Cases: - Test consensus logic without spawning agents - Verify MCP tool calls in isolation - Fast unit tests (<1ms vs 8.7ms real MCP) - Deterministic fixture responses

---

API Reference

Creating a Mock

```
use codex_tui::tests::common::MockMcpManager;

let mut mock = MockMcpManager::new();
```

Methods: - `new()` → Create empty mock - `default()` → Same as `new()` (implements Default)

Adding Fixtures

Single Fixture:

```
mock.add_fixture(
    "local-memory",           // server name
    "search",                 // tool name
    Some("SPEC-TEST plan"),   // query pattern (or None for
wildcard)
    json!({                   // fixture response
        "memory": {
            "id": "test-1",
            "content": "Test content"
            }
        })
    );
```

Multiple Fixtures:

```
mock.add_fixtures(
    "local-memory",
    "search",
    Some("SPEC-TEST plan"),
    vec![
        json!({"memory": {"id": "test-1", "content": "Agent 1"}}),
        json!({"memory": {"id": "test-2", "content": "Agent 2"}}),
    ]
);
```

From File:

```
mock.load_fixture_file(
    "local-memory",
    "search",
    Some("SPEC-KIT-DEMO plan"),
    "tests/fixtures/consensus/demo-plan-gemini.json"
)?;
```

Calling Tools

Signature:

```
pub async fn call_tool(
    &self,
    server: &str,
    tool: &str,
    arguments: Option<Value>,
    timeout: Option<Duration>,
) -> Result<CallToolResult>
```

Example:

```
let args = json!({"query": "SPEC-TEST plan"});
let result = mock.call_tool(
    "local-memory",
    "search",
    Some(args),
    None  // timeout
).await?;

// Extract response
if let ContentBlock::TextContent(text) = &result.content[0] {
    let data: Value = serde_json::from_str(&text.text)?;
    println!("{:?}", data);
}
```

Call Logging

Get Call History:

```
let log = mock.call_log();
for entry in log {
    println!("Called: {}/{}", entry.server, entry.tool);
    println!("  Args: {:?}", entry.arguments);
}
```

Clear Log:

```
mock.clear_log();
```

Use Case: Verify expected tool calls were made

```
assert_eq!(log.len(), 3);
assert_eq!(log[0].tool, "search");
assert_eq!(log[1].tool, "search");
assert_eq!(log[2].tool, "search");
```

Fixture Matching

Priority Order: 1. **Exact query match**: `query_pattern = Some("SPEC-TEST plan")` 2. **Wildcard match**: `query_pattern = None` 3. **No match**: Returns error

Example:

```
// Add wildcard fixture
mock.add_fixture("local-memory", "search", None, json!({"default": true}));

// Add specific fixture
mock.add_fixture(
    "local-memory",
    "search",
    Some("SPEC-DEMO plan"),
    json!({"specific": true})
);

// Query "SPEC-DEMO plan" → Returns {"specific": true}
// Query "anything else"   → Returns {"default": true}
// Query with no fixture   → Error
```

Usage Patterns

Pattern 1: Unit Testing Consensus

```
#[tokio::test]
async fn test_consensus_high_confidence() {
    let mut mock = MockMcpManager::new();

    // Load real production fixtures
    mock.load_fixture_file(
        "local-memory",
        "search",
        Some("SPEC-TEST plan"),
        "tests/fixtures/consensus/demo-plan-gemini.json"
    )?;
    mock.load_fixture_file(
        "local-memory",
        "search",
        Some("SPEC-TEST plan"),
        "tests/fixtures/consensus/demo-plan-claude.json"
    )?;

    // Test consensus collection
    let (results, degraded) = fetch_memory_entries(
        "SPEC-TEST",
        SpecStage::Plan,
        &mock
    ).await?;

    assert_eq!(results.len(), 2);
    assert!(!degraded, "Should have both agents");
}
```

Pattern 2: Verifying Tool Calls

```
#[tokio::test]
async fn test_quality_gate_calls_all_tools() {
    let mut mock = MockMcpManager::new();
    mock.add_fixture("local-memory", "search", None, json!({}));

    // Run quality gate
    run_quality_gate("SPEC-TEST", &mock).await?;

    // Verify calls
    let log = mock.call_log();
    assert!(log.iter().any(|e| e.tool == "search"));

    // Verify call arguments
    let search_call = log.iter().find(|e| e.tool == "search").unwrap();
    assert!(search_call.arguments.is_some());
}
```

Pattern 3: Testing Error Handling

```
#[tokio::test]
async fn test_consensus_degradation_on_missing_agent() {
    let mut mock = MockMcpManager::new();

    // Only add 2 of 3 agents
    mock.add_fixture("local-memory", "search", None, json!({"agent": "gemini"}));
    mock.add_fixture("local-memory", "search", None, json!({"agent": "claude"}));
    // gpt_pro deliberately missing

    let (results, degraded) = fetch_memory_entries(
        "SPEC-TEST",
        SpecStage::Plan,
        &mock
    ).await?;

    assert_eq!(results.len(), 2);
    assert!(degraded, "Should be degraded (missing 1 agent)");
}
```

Tests

Location: `codex-rs/tui/tests/mock_mcp_tests.rs` (7 tests)

Coverage:

```
test_mock_mcp_returns_fixture                    ✓
test_mock_mcp_logs_calls                         ✓
test_mock_mcp_wildcard_matches                   ✓
test_mock_mcp_exact_query_precedence             ✓
test_mock_mcp_multiple_fixtures_return_array     ✓
test_mock_mcp_load_from_file                     ✓
test_mock_mcp_error_on_no_fixture                ✓
```

Run Tests:

```
cd codex-rs
cargo test --test mock_mcp_tests
```

IntegrationTestContext

Purpose

Multi-module test harness for integration tests with isolated filesystem and evidence verification.

Location: `codex-rs/tui/tests/common/integration_harness.rs` (254 LOC)

Use Cases: - Cross-module workflow tests - Evidence verification - Filesystem isolation (temp directories) - SPEC directory structure setup

---

API Reference

Creating a Context

```
use codex_tui::tests::common::IntegrationTestContext;

let ctx = IntegrationTestContext::new("SPEC-TEST-001")?;
```

Fields:

```
pub struct IntegrationTestContext {
    pub temp_dir: TempDir,         // Auto-cleaned on drop
    pub spec_id: String,           // "SPEC-TEST-001"
    pub cwd: PathBuf,              // temp_dir path
    pub evidence_dir: PathBuf,     // docs/SPEC-OPS-004.../evidence
}
```

Auto-Created Directories: - `docs/SPEC-OPS-004-integrated-coder-hooks/evidence/` - `docs/SPEC-OPS-004.../evidence/consensus/{spec_id}/` - `docs/SPEC-OPS-004.../evidence/commands/{spec_id}/`

Directory Helpers

Get Evidence Directories:

```
let consensus_dir = ctx.consensus_dir();
// → .../evidence/consensus/SPEC-TEST-001/

let commands_dir = ctx.commands_dir();
// → .../evidence/commands/SPEC-TEST-001/
```

Create SPEC Directory:

```
let spec_dir = ctx.create_spec_dirs("test-feature")?;
// → .../docs/SPEC-TEST-001-test-feature/
```

File Helpers

Write PRD:

```
ctx.write_prd("test-feature", "# PRD\n\nTest product requirements")?;
// Creates: docs/SPEC-TEST-001-test-feature/PRD.md
```

Write Spec:

```
ctx.write_spec("test-feature", "# SPEC-TEST-001\n\n## Goal\nTest")?;
// Creates: docs/SPEC-TEST-001-test-feature/spec.md
```

Evidence Verification

Check Consensus Artifacts:

```
// Single agent
let exists = ctx.assert_consensus_exists(SpecStage::Plan, "gemini");
assert!(exists);

// All agents (via EvidenceVerifier)
let verifier = EvidenceVerifier::new(&ctx);
assert!(verifier.assert_consensus_complete(
    SpecStage::Plan,
    &["gemini", "claude", "gpt_pro"]
));
```

Check Guardrail Telemetry:

```
let exists = ctx.assert_guardrail_telemetry_exists(SpecStage::Plan);
assert!(exists);
```

Count Files:

```
let count = ctx.count_consensus_files();
assert_eq!(count, 3, "Should have 3 agent outputs");

let guardrail_count = ctx.count_guardrail_files();
assert_eq!(guardrail_count, 1, "Should have 1 telemetry file");
```

Usage Patterns

Pattern 1: Workflow Integration Test

```
        #[tokio::test]
        async fn test_full_plan_stage_workflow() -> Result<()> {
            // Setup
            let ctx = IntegrationTestContext::new("SPEC-INT-001")?;
            ctx.write_prd("test-feature", "# Test PRD\n\n## Goal\nTest")?;

            // Run plan stage
            run_plan_stage(&ctx.spec_id, &ctx.cwd).await?;

            // Verify evidence
            assert!(ctx.assert_consensus_exists(SpecStage::Plan, "gemini"));
            assert!(ctx.assert_consensus_exists(SpecStage::Plan, "claude"));
            assert!(ctx.assert_consensus_exists(SpecStage::Plan,
"gpt_pro"));
            assert!(ctx.assert_guardrail_telemetry_exists(SpecStage::Plan));

            // Verify file count
            assert_eq!(ctx.count_consensus_files(), 3);

            Ok(())
        }
```

Pattern 2: Error Recovery Test

```
        #[tokio::test]
        async fn test_error_recovery_creates_evidence() -> Result<()> {
            let ctx = IntegrationTestContext::new("SPEC-INT-002")?;

            // Simulate error (missing PRD)
            let result = run_plan_stage(&ctx.spec_id, &ctx.cwd).await;
            assert!(result.is_err());

            // Verify error evidence still created
            let verifier = EvidenceVerifier::new(&ctx);
            assert!
(verifier.assert_guardrail_valid(SpecStage::Plan).is_ok());

            Ok(())
        }
```

Pattern 3: State Persistence Test

```
        #[tokio::test]
        async fn test_state_persists_across_stages() -> Result<()> {
            let ctx = IntegrationTestContext::new("SPEC-INT-003")?;
            ctx.write_prd("test", "# PRD")?;

            // Run plan
            run_plan_stage(&ctx.spec_id, &ctx.cwd).await?;
            assert_eq!(ctx.count_consensus_files(), 3);

            // Run tasks (should accumulate, not replace)
            run_tasks_stage(&ctx.spec_id, &ctx.cwd).await?;
            assert!(ctx.count_consensus_files() > 3, "Should accumulate
evidence");

            Ok(())
        }
```

Tests

Location: `codex-rs/tui/tests/common/integration_harness.rs` (4 tests in `mod tests`)

Coverage:

```
test_integration_context_creation    ✓
test_state_builder                   ✓
test_spec_dirs_creation              ✓
test_evidence_verifier               ✓
```

StateBuilder

Purpose

Builder pattern for creating `SpecAutoState` instances in tests with custom configuration.

Location: `codex-rs/tui/tests/common/integration_harness.rs`

Use Cases: - Configure test automation state - Test different starting stages - Test HAL mode variations - Test quality gate configurations

---

API Reference

Basic Usage

```
use codex_tui::tests::common::StateBuilder;

let state = StateBuilder::new("SPEC-TEST-001").build();
```

Default Configuration: - `goal`: "Integration test" - `start_stage`: Plan - `hal_mode`: None - `quality_gates_enabled`: true

---

Builder Methods

Custom Goal:

```
let state = StateBuilder::new("SPEC-TEST-001")
    .with_goal("Implement user authentication")
    .build();
```

Start at Different Stage:

```
let state = StateBuilder::new("SPEC-TEST-002")
    .starting_at(SpecStage::Implement)
    .build();
```

HAL Mode Configuration:

```
let state = StateBuilder::new("SPEC-TEST-003")
    .with_hal_mode(HalMode::Analyze)
    .build();
```

Quality Gates Control:

```
let state = StateBuilder::new("SPEC-TEST-004")
    .quality_gates(false)  // Disable quality gates
    .build();
```

Chained Configuration:

```
let state = StateBuilder::new("SPEC-TEST-005")
    .with_goal("Test refactoring")
    .starting_at(SpecStage::Validate)
    .with_hal_mode(HalMode::TestOnly)
    .quality_gates(true)
    .build();
```

Usage Patterns

Pattern 1: Testing Stage Transitions

```
#[test]
fn test_stage_advancement() {
    let mut state = StateBuilder::new("SPEC-TEST-001")
        .starting_at(SpecStage::Plan)
        .build();

    assert_eq!(state.current_stage(), Some(SpecStage::Plan));

    state.advance_stage();
    assert_eq!(state.current_stage(), Some(SpecStage::Tasks));

    state.advance_stage();
    assert_eq!(state.current_stage(), Some(SpecStage::Implement));
}
```

Pattern 2: Testing Quality Gate Behavior

```
#[test]
fn test_quality_gates_disabled() {
    let state = StateBuilder::new("SPEC-TEST-002")
        .quality_gates(false)
        .build();

    assert!(!state.quality_gates_enabled);

    // Quality gates should not run
    assert!(should_skip_quality_gate(&state));
}

#[test]
fn test_quality_gates_enabled() {
    let state = StateBuilder::new("SPEC-TEST-003")
        .quality_gates(true)
        .build();

    assert!(state.quality_gates_enabled);
}
```

Pattern 3: Testing HAL Integration

```
#[test]
fn test_hal_mode_analyze() {
    let state = StateBuilder::new("SPEC-TEST-004")
        .with_hal_mode(HalMode::Analyze)
        .build();

    assert_eq!(state.hal_mode, Some(HalMode::Analyze));
}

#[test]
fn test_hal_mode_none() {
    let state = StateBuilder::new("SPEC-TEST-005")
        .build();

    assert_eq!(state.hal_mode, None);
}
```

EvidenceVerifier

Purpose

Helper for verifying evidence artifacts in integration tests.

Location: `codex-rs/tui/tests/common/integration_harness.rs`

Use Cases: - Assert consensus artifacts exist - Validate guardrail telemetry - Verify directory structure - Check multi-agent completion

---

API Reference

Creating a Verifier

```
use codex_tui::tests::common::EvidenceVerifier;

let ctx = IntegrationTestContext::new("SPEC-TEST-001")?;
let verifier = EvidenceVerifier::new(&ctx);
```

Verification Methods

Consensus Complete (all agents present):

```
let complete = verifier.assert_consensus_complete(
    SpecStage::Plan,
    &["gemini", "claude", "gpt_pro"]
);
assert!(complete);
```

Guardrail Valid (telemetry exists and parseable):

```
let result = verifier.assert_guardrail_valid(SpecStage::Plan);
assert!(result.is_ok());
```

Structure Valid (directories exist):

```
let valid = verifier.assert_structure_valid();
assert!(valid);
```

Usage Patterns

Pattern 1: Post-Workflow Verification

```
#[tokio::test]
async fn test_plan_creates_complete_evidence() -> Result<()> {
    let ctx = IntegrationTestContext::new("SPEC-VER-001")?;
    ctx.write_prd("test", "# PRD")?;

    run_plan_stage(&ctx.spec_id, &ctx.cwd).await?;

    let verifier = EvidenceVerifier::new(&ctx);

    // Verify all artifacts
    assert!(verifier.assert_structure_valid());
    assert!(verifier.assert_consensus_complete(
        SpecStage::Plan,
        &["gemini", "claude", "gpt_pro"]
    ));
    assert!
(verifier.assert_guardrail_valid(SpecStage::Plan).is_ok());

    Ok(())
}
```

Pattern 2: Degraded Consensus Detection

```
#[tokio::test]
async fn test_degraded_consensus_still_valid() -> Result<()> {
    let ctx = IntegrationTestContext::new("SPEC-VER-002")?;

    // Simulate degraded consensus (only 2/3 agents)
    simulate_agent_failure("gpt_pro")?;
    run_plan_stage(&ctx.spec_id, &ctx.cwd).await?;

    let verifier = EvidenceVerifier::new(&ctx);

    // Should NOT be complete (missing 1 agent)
    assert!(!verifier.assert_consensus_complete(
        SpecStage::Plan,
        &["gemini", "claude", "gpt_pro"]
    ));

    // But 2/3 is still valid
    assert!(verifier.assert_consensus_complete(
        SpecStage::Plan,
        &["gemini", "claude"]
    ));

    Ok(())
}
```

Fixture Library

Overview

Location: `codex-rs/tui/tests/fixtures/consensus/` (20 files, 96 KB)

Source: Real production artifacts from `docs/SPEC-OPS-004.../evidence/consensus/`

Coverage: - Plan stage: 13 fixtures (DEMO, 025, 045) - Tasks stage: 3 fixtures (025) - Implement stage: 4 fixtures (025)

File Naming Convention

Format: `{spec_id}-{stage}-{agent}.json`

Examples: - `demo-plan-gemini.json` — SPEC-KIT-DEMO plan stage (Gemini output) - `025-implement-gpt_codex.json` — SPEC-KIT-025 implement stage (Codex output) - `045-plan-claude.json` — SPEC-KIT-045 plan stage (Claude output)

Available Fixtures

Plan Stage (13 files)

SPEC-KIT-DEMO: - `demo-plan-gemini.json` (14 KB) - `demo-plan-claude.json` (12 KB) - `demo-plan-gpt_pro.json` (15 KB)

SPEC-KIT-025 (Native SPEC-ID generation): - `025-plan-gemini.json` (16 KB) - `025-plan-claude.json` (14 KB) - `025-plan-gpt_pro.json` (18 KB)

SPEC-KIT-045 (Quality gate handler): - `045-plan-gemini.json` (13 KB) - `045-plan-claude.json` (11 KB) - `045-plan-gpt_pro.json` (17 KB)

Tasks Stage (3 files)

SPEC-KIT-025: - `025-tasks-gemini.json` (8 KB) - `025-tasks-claude.json` (7 KB)

Implement Stage (4 files)

SPEC-KIT-025: - `025-implement-gemini.json` (9 KB) - `025-implement-claude.json` (8 KB) - `025-implement-gpt_codex.json` (22 KB) — Code implementation - `025-implement-gpt_pro.json` (11 KB)

Usage in Tests

Loading Single Fixture:

```
let mut mock = MockMcpManager::new();
mock.load_fixture_file(
    "local-memory",
    "search",
    Some("SPEC-KIT-DEMO plan"),
    "tests/fixtures/consensus/demo-plan-gemini.json"
)?;
```

Loading All Agents (simulate 3-agent consensus):

```
let mut mock = MockMcpManager::new();
let agents = vec!["gemini", "claude", "gpt_pro"];

for agent in agents {
    mock.load_fixture_file(
        "local-memory",
        "search",
        Some("SPEC-KIT-DEMO plan"),
        &format!("tests/fixtures/consensus/demo-plan-{}.json", agent)
    )?;
}
```

Loading Different Stages:

```
// Plan stage
mock.load_fixture_file("local-memory", "search", Some("SPEC-KIT-025 plan"),
    "tests/fixtures/consensus/025-plan-gemini.json")?;

// Tasks stage
mock.load_fixture_file("local-memory", "search", Some("SPEC-KIT-025 tasks"),
    "tests/fixtures/consensus/025-tasks-gemini.json")?;

// Implement stage
mock.load_fixture_file("local-memory", "search", Some("SPEC-KIT-025 implement"),
    "tests/fixtures/consensus/025-implement-gpt_codex.json")?;
```

Adding New Fixtures

Manual Creation:

```
cd codex-rs/tui/tests/fixtures/consensus

# Copy from production evidence
cp ../../../docs/SPEC-OPS-004.../evidence/consensus/SPEC-KIT-070/spec-plan_*.json \
   ./070-plan-gemini.json
```

Automated Extraction (future):

```
# Extract fixtures from evidence repository
./scripts/extract_test_fixtures.sh SPEC-KIT-070
```

Size Guidelines: - Keep individual fixtures < 30 KB - Total fixture directory < 200 KB - Compress if needed (not implemented yet)

Coverage Tools

cargo-tarpaulin

Purpose: Line coverage measurement for Rust code

Installation:

```
cargo install cargo-tarpaulin
```

Configuration: `codex-rs/tarpaulin.toml`

Configuration Details

```
[config]
# Only measure spec-kit coverage (fork-specific code)
run-types = ["Lib", "Tests"]

# Include patterns (spec-kit only)
include-pattern = "tui/src/chatwidget/spec_kit/.*\\.rs"

# Exclude test files and generated code
exclude-files = [
    "tui/src/chatwidget/spec_kit/*/tests/*",
    "tui/tests/*",
]

# Output formats
out = ["Html", "Stdout"]
output-dir = "target/tarpaulin"

# Timeout per test (integration tests are slow)
timeout = 120

# Verbose output
verbose = true
```

Usage

Full Coverage Report:

```
cd codex-rs
cargo tarpaulin
```

Output:

```
|| Tested/Total Lines:
|| tui/src/chatwidget/spec_kit/handler.rs: 145/961
|| tui/src/chatwidget/spec_kit/consensus.rs: 120/992
|| tui/src/chatwidget/spec_kit/quality.rs: 178/807
|| ...
||
|| Coverage: 42.3%
```

Specific Module:

```
cargo tarpaulin -p codex-tui
```

HTML Report:

```
cargo tarpaulin --out Html
```

```
open target/tarpaulin/index.html
```

XML for CI (Codecov):

```
cargo tarpaulin --out Xml
```

Troubleshooting

Issue: Timeout on slow tests

```
# Increase timeout
cargo tarpaulin --timeout 300
```

Issue: Out of memory

```
# Reduce parallelism
cargo tarpaulin --jobs 2
```

Issue: Incorrect coverage (too low)

```
# Ensure all features enabled
cargo tarpaulin --all-features
```

---

cargo-llvm-cov

Purpose: Alternative coverage tool using LLVM instrumentation

Advantages: - More accurate than tarpaulin - Faster execution - Better integration with IDEs

Installation:

```
cargo install cargo-llvm-cov
```

Usage

Generate Coverage:

```
cd codex-rs
cargo llvm-cov --workspace --all-features --html
```

Open Report:

```
open target/llvm-cov/html/index.html
```

JSON Output (for parsing):

```
cargo llvm-cov --workspace --all-features --json --output-path coverage.json
```

Integration with VS Code:

```
# Install Coverage Gutters extension
# Run:
cargo llvm-cov --workspace --all-features --lcov --output-path lcov.info

# VS Code will show coverage inline
```

Comparison: Tarpaulin vs llvm-cov

Feature	Tarpaulin	llvm-cov
Accuracy	~95%	~99%
Speed	Baseline	1.5-2× faster
HTML Report	✅ Good	✅ Excellent
IDE Integration	❌ Limited	✅ VS Code, IntelliJ
CI Support	✅ Codecov, Coveralls	✅ All platforms
Install Size	50 MB	150 MB (LLVM)

Recommendation: Use llvm-cov for local development, tarpaulin for CI (smaller install).

Property-Based Testing

Overview

Purpose: Generative testing with random inputs to verify invariants

Tool: proptest (Rust equivalent of Hypothesis/QuickCheck)

Location: `codex-rs/tui/tests/property_based_tests.rs`

Use Cases: - State machine invariants - Evidence integrity - Consensus edge cases - Input validation

---

Proptest Basics

Simple Property Test:

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_state_index_never_negative(index in 0usize..20) {
        // Property: State always handles any index gracefully
        let mut state = SpecAutoState::new(...);
        state.current_index = index;

        // Should never panic
        let _ = state.current_stage();
    }
}
```

How It Works: 1. Generate 100 random values for `index` (0-19) 2. Run test with each value 3. If any fails, shrink to minimal failing case 4. Report failure with minimal input

---

Test Categories

PB01-PB03: State Invariants

PB01: Index always in valid range

```
proptest! {
    #[test]
    fn pb01_state_index_always_in_valid_range(index in 0usize..20) {
        let mut state = StateBuilder::new("SPEC-PB01-TEST")
            .starting_at(SpecStage::Plan)
            .build();

        state.current_index = index;

        // Invariant: index ∈ [0, 5] → Some(_), else None
        if index < 6 {
            prop_assert!(state.current_stage().is_some());
        } else {
            prop_assert_eq!(state.current_stage(), None);
        }
    }
}
```

PB02: Current stage always Some when index < 6

```
proptest! {
    #[test]
    fn pb02_current_stage_always_some_when_index_under_six(
        index in 0usize..6
    ) {
        let mut state = StateBuilder::new("SPEC-PB02-TEST").build();
        state.current_index = index;

        prop_assert!(state.current_stage().is_some());
    }
}
```

PB03: Retry count never exceeds max

```
proptest! {
    #[test]
    fn pb03_retry_count_never_negative(retries in 0usize..100) {
        let max_retries = 3;
        let capped_retries = retries.min(max_retries);

        // Write retry file
        let retry_data = json!({
            "retry_count": capped_retries,
            "max_retries": max_retries,
        });

        // Invariant: retry_count ≤ max_retries
        prop_assert!(capped_retries <= max_retries);
    }
}
```

PB04-PB06: Evidence Integrity

PB04: Written evidence always parseable JSON

```
proptest! {
    #[test]
    fn pb04_written_evidence_always_parseable_json(
        agent in "[a-z]{3,10}",
        content in ".*"
    ) {
        let ctx = IntegrationTestContext::new("SPEC-PB04-TEST")?;

        let evidence = json!({
            "agent": agent,
            "content": content,
            "timestamp": "2025-10-19T00:00:00Z"
        });

        let file = ctx.consensus_dir().join("test.json");
        std::fs::write(&file, evidence.to_string())?;

        // Invariant: File is valid JSON
        let content = std::fs::read_to_string(&file)?;
        let parsed: Value = serde_json::from_str(&content)?;

        prop_assert_eq!(parsed["agent"].as_str(), Some(agent.as_str()));
    }
}
```

Custom Generators

Generate SPEC IDs:

```
fn spec_id_strategy() -> impl Strategy<Value = String> {
    "[A-Z]{4}-[A-Z]{3}-[0-9]{3}"
        .prop_map(|s| s.to_string())
}

proptest! {
    #[test]
    fn test_spec_id_parsing(spec_id in spec_id_strategy()) {
        // Test SPEC ID validation
        assert!(is_valid_spec_id(&spec_id));
    }
}
```

Generate Stages:

```
fn stage_strategy() -> impl Strategy<Value = SpecStage> {
    prop_oneof![
        Just(SpecStage::Plan),
        Just(SpecStage::Tasks),
        Just(SpecStage::Implement),
        Just(SpecStage::Validate),
        Just(SpecStage::Audit),
        Just(SpecStage::Unlock),
    ]
}
```

Running Property Tests

Run All Property Tests:

```
cd codex-rs
```

```
cargo test --test property_based_tests
```

Run Specific Test:

```
cargo test --test property_based_tests pb01_state_index
```

Adjust Iteration Count (default 100):

```
PROPTEST_CASES=1000 cargo test --test property_based_tests
```

Debug Failing Case:

```
# proptest creates a regression file
cat proptest-regressions/property_based_tests.txt

# Re-run with that specific input
cargo test --test property_based_tests -- --exact pb01_state_index
```

TestCodexBuilder

Purpose

Builder for creating test instances of `CodexConversation` with mock servers.

Location: `codex-rs/core/tests/common/test_codex.rs` (76 LOC)

Use Cases: - Test agent spawning - Test conversation lifecycle - Test configuration variations - Integration with wiremock

---

API Reference

Basic Usage:

```
use codex_core::tests::common::test_codex;

let server = wiremock::MockServer::start().await;
let codex = test_codex()
    .build(&server)
    .await?;
```

Fields:

```
pub struct TestCodex {
    pub home: TempDir,                         // Isolated home directory
    pub cwd: TempDir,                          // Isolated working directory
    pub codex: Arc<CodexConversation>,         // Conversation instance
    pub session_configured: SessionConfiguredEvent,  // Initial event
}
```

Custom Configuration

Modify Config:

```
let codex = test_codex()
    .with_config(|config| {
        config.model = "gpt-5-low".to_string();
        config.max_tokens = 4096;
    })
    .build(&server)
    .await?;
```

Multiple Mutations:

```
let codex = test_codex()
    .with_config(|config| config.model = "gpt-5-low".to_string())
    .with_config(|config| config.max_tokens = 8192)
    .with_config(|config| config.temperature = 0.7)
    .build(&server)
    .await?;
```

Usage with Wiremock

Mock API Responses:

```
use wiremock::{MockServer, Mock, ResponseTemplate};
use wiremock::matchers::{method, path};

#[tokio::test]
async fn test_conversation_with_mock() -> Result<()> {
    let server = MockServer::start().await;

    // Mock /v1/chat/completions
    Mock::given(method("POST"))
        .and(path("/v1/chat/completions"))
        .respond_with(ResponseTemplate::new(200).set_body_json(json!({
            "id": "chatcmpl-test",
            "object": "chat.completion",
            "created": 1234567890,
            "model": "gpt-4",
            "choices": [{
                "index": 0,
                "message": {
                    "role": "assistant",
                    "content": "Test response"
                },
                "finish_reason": "stop"
            }]
        })))
        .mount(&server)
        .await;

    let codex = test_codex().build(&server).await?;

    // Test conversation
    let response = codex.codex.send_message("Test").await?;
    assert_eq!(response.content, "Test response");

    Ok(())
}
```

Common Test Utilities

Test Module Structure

Location: `codex-rs/tui/tests/common/mod.rs`

```rust
//! Common test utilities for spec-kit

pub mod integration_harness;
pub mod mock_mcp;

pub use integration_harness::{
    EvidenceVerifier,
    IntegrationTestContext,
    StateBuilder,
};
pub use mock_mcp::MockMcpManager;
```

Usage in Tests:

```rust
mod common;

use common::{
    MockMcpManager,
    IntegrationTestContext,
    StateBuilder,
    EvidenceVerifier,
};
```

Shared Test Data

Constants:

```rust
// tests/common/mod.rs

pub const TEST_SPEC_ID: &str = "SPEC-TEST-001";
pub const TEST_GOAL: &str = "Integration test";

pub fn default_test_prd() -> &'static str {
    r#"
# Product Requirements Document

## Goal
Test feature implementation

## Requirements
- R1: Feature should work
- R2: Feature should be tested
    "#
}
```

Usage:

```rust
use common::{TEST_SPEC_ID, default_test_prd};

#[tokio::test]
async fn test_with_shared_data() {
    let ctx = IntegrationTestContext::new(TEST_SPEC_ID)?;
    ctx.write_prd("test-feature", default_test_prd())?;
    // ...
}
```

Test Organization Best Practices

File Naming

Unit Tests (in source files):

```
// src/chatwidget/spec_kit/handler.rs

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_handler_orchestration() { }
}
```

Integration Tests (separate files):

```
codex-rs/tui/tests/
├── workflow_integration_tests.rs
├── error_recovery_integration_tests.rs
├── state_persistence_integration_tests.rs
├── concurrent_operations_integration_tests.rs
└── quality_flow_integration_tests.rs
```

Property Tests:

```
codex-rs/tui/tests/property_based_tests.rs
```

Test Naming Conventions

Pattern: `test_{what}_{condition}_{expected}`

Examples:

```
#[test]
fn test_state_advance_increments_index() { }

#[test]
fn test_consensus_degraded_when_missing_agent() { }

#[test]
fn test_evidence_created_on_error() { }

#[tokio::test]
async fn test_quality_gate_passes_when_score_above_80() { }
```

Avoid:

```
#[test]
fn test1() { }  // ❌ Meaningless
```

```rust
#[test]
fn it_works() { }  // ✖ Too vague
```

Common Test Patterns

Pattern: Arrange-Act-Assert

```rust
#[test]
fn test_example() {
    // Arrange: Setup
    let ctx = IntegrationTestContext::new("SPEC-TEST")?;
    let state = StateBuilder::new("SPEC-TEST").build();

    // Act: Execute
    let result = do_something(&ctx, &state)?;

    // Assert: Verify
    assert_eq!(result, expected);
}
```

Pattern: Given-When-Then

```rust
#[tokio::test]
async fn test_consensus_with_degradation() {
    // Given: 3-agent consensus with 1 agent failing
    let mut mock = MockMcpManager::new();
    mock.add_fixture("local-memory", "search", None, json!({"agent": "gemini"}));
    mock.add_fixture("local-memory", "search", None, json!({"agent": "claude"}));
    // gpt_pro missing (simulates failure)

    // When: Fetch consensus
    let (results, degraded) = fetch_memory_entries(
        "SPEC-TEST",
        SpecStage::Plan,
        &mock
    ).await?;

    // Then: Should have 2/3 agents and be degraded
    assert_eq!(results.len(), 2);
    assert!(degraded);
}
```

Pattern: Table-Driven Tests

```rust
#[test]
fn test_stage_index_mapping() {
    let test_cases = vec![
        (0, Some(SpecStage::Plan)),
        (1, Some(SpecStage::Tasks)),
        (2, Some(SpecStage::Implement)),
        (3, Some(SpecStage::Validate)),
        (4, Some(SpecStage::Audit)),
        (5, Some(SpecStage::Unlock)),
        (6, None),
    ];

    for (index, expected) in test_cases {
        let mut state = StateBuilder::new("SPEC-TEST").build();
        state.current_index = index;
        assert_eq!(state.current_stage(), expected);
    }
}
```

Summary

Test Infrastructure Highlights:

1. **MockMcpManager**: Fixture-based MCP testing (272 LOC, 7 tests)
2. **IntegrationTestContext**: Isolated filesystem, evidence verification
3. **StateBuilder**: Test state configuration with fluent API
4. **EvidenceVerifier**: Artifact validation helpers
5. **Fixture Library**: 20 real production artifacts (96 KB)
6. **Coverage Tools**: cargo-tarpaulin (CI), cargo-llvm-cov (local)
7. **Property Testing**: proptest for generative invariant testing
8. **TestCodexBuilder**: Conversation mocking with wiremock

Benefits: - ✅ Fast tests (no external dependencies) - ✅ Deterministic (fixture-based) - ✅ Isolated (temp directories) - ✅ Comprehensive (unit, integration, property) - ✅ Measurable (coverage tools)

Next Steps: - Unit Testing Guide - Writing effective unit tests - Integration Testing Guide - Cross-module tests - Property Testing Guide - Generative testing patterns

---

References: - MockMcpManager: `codex-rs/tui/tests/common/mock_mcp.rs` - IntegrationTestContext: `codex-rs/tui/tests/common/integration_harness.rs` - Tarpaulin config: `codex-rs/tarpaulin.toml` - Property tests: `codex-rs/tui/tests/property_based_tests.rs` - TestCodexBuilder: `codex-rs/core/tests/common/test_codex.rs`

Testing Strategy

Comprehensive testing approach for the codebase.

Overview

Testing Philosophy: Balance coverage, confidence, and development velocity

Current Metrics (as of 2025-11-17): - **Total Tests**: 604 tests across all modules - **Pass Rate**: 100% (all tests passing) - **Coverage**: 42-48% (estimated, varies by module) - **Target**: 40%+ coverage minimum

Test Distribution: - **Unit Tests**: ~380 tests (63%) - **Integration Tests**: ~200 tests (33%) - **E2E Tests**: ~24 tests (4%)

Location: Tests located alongside source in `tests/` directories per module

---

Coverage Goals

Overall Target: 40%+

Rationale: - Industry standard for Rust projects: 60-80% - Our target: 40%+ given complexity and time constraints - Current achievement: 42-48% ✅ **Target Met**

Coverage by Priority: - **Critical paths**: 70-80% (Spec-Kit automation, MCP client) - **Core functionality**: 50-60% (TUI, database, config) - **Supporting code**: 30-40% (utilities, helpers) - **Legacy code**: 20-30% (minimal coverage acceptable)

Module-Specific Targets

Module	Priority	Target Coverage	Current Est.	Status
codex-tui/spec_kit	Critical	70%	~75%	✅ Exceeded
codex-mcp-client	Critical	70%	~65%	🔄 Near target
codex-tui	High	50%	~45%	🔄 Near target
codex-core	High	50%	~50%	✅ Met
codex-db	High	50%	~60%	✅ Exceeded
config-loader	Medium	40%	~55%	✅ Exceeded
file-search	Medium	40%	~40%	✅ Met
utilities	Low	30%	~35%	✅ Met

Overall Status: ✅ **42-48% coverage achieved** (exceeds 40% target)

Testing Pyramid

Level 1: Unit Tests (~63%)

Purpose: Test individual functions/components in isolation

Characteristics: - Fast execution (<1s for all unit tests) - No external dependencies (mocked) - High volume (~380 tests)

What to Unit Test: - ✅ Pure functions (input → output, no side effects) - ✅ Business logic (validation, parsing, calculations) - ✅ Data structures (serialization, deserialization) - ✅ Error handling (edge cases, invalid inputs)

What NOT to Unit Test: - ❌ Integration points (use integration tests) - ❌ UI rendering (hard to test, low ROI) - ❌ External APIs (mock in integration tests)

Example Coverage:

```
spec_kit/clarify_native.rs: 85% (pattern matching logic)
spec_kit/checklist_native.rs: 90% (scoring algorithms)
mcp-client/protocol.rs: 75% (JSON-RPC parsing)
```

Level 2: Integration Tests (~33%)

Purpose: Test multiple modules working together

Characteristics: - Moderate execution time (1-10s per test) - Real module interactions (no mocks between modules) - Medium volume (~200 tests)

What to Integration Test: - ✅ Workflow orchestration (plan → tasks → implement) - ✅ Cross-module communication (TUI ↔ MCP client) - ✅ State persistence (database writes/reads) - ✅ Error propagation across modules

Example Coverage:

```
spec_kit/workflow_integration_tests.rs: 60 tests
mcp_client/integration_tests.rs: 45 tests
database/integration_tests.rs: 40 tests
```

Level 3: E2E Tests (~4%)

Purpose: Test complete user workflows end-to-end

Characteristics: - Slow execution (10-60s per test) - Full stack (TUI + backend + database + MCP) - Low volume (~24 tests, high value)

What to E2E Test: - ✅ Critical user journeys (`/speckit.auto` full pipeline) - ✅ Error recovery (retry logic, degradation) - ✅ Tmux session management - ✅ Configuration hot-reload

Example Coverage:

```
spec_kit/e2e_tests.rs: 12 tests (full automation)
tmux/e2e_tests.rs: 8 tests (session lifecycle)
config/e2e_tests.rs: 4 tests (hot-reload)
```

Test Organization

Per-Module Tests

Structure:

```
codex-rs/
├── tui/
│   ├── src/
│   │   └── chatwidget/
│   │       └── spec_kit/
│   │           ├── clarify_native.rs
│   │           └── mod.rs
│   └── tests/
│       └── spec_kit/
│           ├── clarify_native_tests.rs      (unit)
│           ├── workflow_integration_tests.rs (integration)
│           └── e2e_tests.rs                  (E2E)
```

Naming Conventions: - Unit tests: `{module}_tests.rs` or `#[cfg(test)] mod tests` in source - Integration tests: `{feature}_integration_tests.rs` - E2E tests: `e2e_tests.rs` or `{workflow}_e2e.rs`

Workspace-Level Tests

Location: `codex-rs/tests/` (workspace root)

Purpose: Cross-crate integration tests

Example:

```
codex-rs/tests/
├── tui_mcp_integration.rs     # TUI ↔ MCP client integration
├── full_pipeline_e2e.rs       # Complete /speckit.auto workflow
└── hot_reload_integration.rs  # Config changes across crates
```

Coverage Measurement

Tools

Primary: `cargo-tarpaulin`

Installation:

```
cargo install cargo-tarpaulin
```

Usage:

```
# All modules
cargo tarpaulin --workspace --all-features --timeout 300

# Specific module
cargo tarpaulin -p codex-tui --all-features

# HTML report
cargo tarpaulin --workspace --all-features --out Html
```

Configuration (`.tarpaulin.toml`):

```
[tarpaulin]
timeout = "300s"
exclude-files = [
    "target/*",
    "*/tests/*",
    "*/benches/*"
]
```

Alternative: `cargo-llvm-cov`

Installation:

```
cargo install cargo-llvm-cov
```

Usage:

```
# Generate coverage
cargo llvm-cov --workspace --all-features --html

# Open report
open target/llvm-cov/html/index.html
```

Advantage: More accurate than tarpaulin, faster execution

Critical Path Coverage

Priority 1: Spec-Kit Automation (70%+ target)

Critical Flows: 1. `/speckit.new` → SPEC creation 2. `/speckit.auto` → Full 6-stage pipeline 3. Quality gates → Checkpoint validation 4. Consensus → Multi-agent synthesis

Current Coverage: ~75% ✅

Key Test Files:

```
tui/tests/spec_kit/
├── new_native_tests.rs              (95 tests)
├── pipeline_coordinator_tests.rs    (85 tests)
├── quality_gate_handler_tests.rs    (75 tests)
├── consensus_coordinator_tests.rs   (45 tests)
└── workflow_integration_tests.rs    (60 tests)
```

Priority 2: MCP Client (70%+ target)

Critical Flows: 1. JSON-RPC protocol → Serialization/deserialization 2. Connection lifecycle → Connect, request, disconnect 3. Tool invocation → MCP tool calls 4. Error handling → Retry logic, timeouts

Current Coverage: ~65% 🔄

Key Test Files:

```
mcp-client/tests/
├── protocol_tests.rs        (40 tests)
├── connection_tests.rs      (30 tests)
├── tool_invocation_tests.rs    (25 tests)
└── integration_tests.rs        (45 tests)
```

Priority 3: Database Layer (50%+ target)

Critical Flows: 1. Schema migrations → Up/down migrations 2. CRUD operations → Insert, query, update, delete 3. Connection pooling → R2D2 integration 4. Transaction handling → Rollback on error

Current Coverage: ~60% ✅

Key Test Files:

```
db/tests/
├── schema_tests.rs         (20 tests)
├── crud_tests.rs           (35 tests)
├── pool_tests.rs           (15 tests)
└── transaction_tests.rs    (10 tests)
```

Test Execution Strategy

Local Development

Run all tests:

```
cd codex-rs
cargo test --workspace --all-features
```

Run specific module:

```
cargo test -p codex-tui --all-features
```

Run specific test:

```
cargo test -p codex-tui
spec_kit::clarify_native::tests::detect_vague_language
```

Run with output:

```
cargo test -- --nocapture
```

Pre-Commit Hook

Location: `.githooks/pre-commit`

What it runs:

```
# Format check
cargo fmt --all -- --check

# Linting
cargo clippy --workspace --all-targets --all-features -- -D warnings

# Quick test (compilation only, no execution)
cargo test --workspace --no-run
```

Time: ~30 seconds (fast feedback)

Skip (if needed):

```
PRECOMMIT_FAST_TEST=0 git commit -m "..."
```

Pre-Push Hook

Location: `.githooks/pre-push`

What it runs:

```
# Format check
cargo fmt --all -- --check

# Linting
cargo clippy --workspace --all-targets --all-features -- -D warnings

# Build
cargo build --workspace --all-features

# Optional: Full test suite (slow)
# cargo test --workspace --all-features
```

Time: ~2-5 minutes

Skip (if needed):

```
PREPUSH_FAST=0 git push
```

CI/CD Pipeline

Location: `.github/workflows/rust.yml`

Triggers: - Push to `main` - Pull requests - Manual workflow dispatch

Jobs: 1. **Test** (parallel matrix): - OS: Ubuntu, macOS, Windows - Rust: stable, beta - Features: all, default

2. **Coverage** (Ubuntu only):
   - Run `cargo-tarpaulin`
   - Upload to Codecov
   - Comment PR with coverage delta
3. **Lint**:
   - `cargo fmt --check`
   - `cargo clippy -- -D warnings`

Time: ~10-15 minutes total

Coverage Gaps

Known Gaps (Acceptable)

UI Rendering (~10% coverage): - **Reason**: Ratatui rendering hard to test - **Mitigation**: Manual testing, visual inspection

Error Handling Paths (~30% coverage): - **Reason**: Hard to trigger rare errors - **Mitigation**: Property-based testing (proptest)

Legacy Code (~20% coverage): - **Reason**: Technical debt, low ROI - **Mitigation**: Refactor on touch, add tests incrementally

---

Prioritized Improvements

Phase 1 (Completed): 40%+ coverage - ✅ Spec-Kit core functionality (360 tests added) - ✅ MCP client protocol (140 tests added) - ✅ Database layer (80 tests added)

Phase 2 (Optional): 50%+ coverage - 🔄 Error recovery scenarios - 🔄 Concurrent operation tests - 🔄 Edge case property testing

Phase 3 (Future): 60%+ coverage - ⏳ UI interaction tests - ⏳ Performance regression tests - ⏳ Chaos engineering tests

---

Testing Best Practices

DO

✅ **Test behavior, not implementation**:

```rust
// Good: Test behavior
#[test]
fn clarify_detects_vague_language() {
    let result = clarify("System should be fast");
    assert!(result.has_ambiguities());
    assert_eq!(result.ambiguities[0].pattern, "vague_language");
}

// Bad: Test implementation details
#[test]
fn clarify_calls_regex_find() {
    // Don't test internal regex usage
}
```

✅ **Use descriptive test names**:

```rust
#[test]
fn checklist_fails_when_score_below_80() { }

#[test]
fn consensus_degraded_when_only_2_of_3_agents() { }
```

✅ **Arrange-Act-Assert pattern**:

```rust
#[test]
fn test_feature() {
    // Arrange: Setup
    let input = "test input";

    // Act: Execute
    let result = function_under_test(input);

    // Assert: Verify
    assert_eq!(result, expected);
}
```

DON’T

❌ **Test framework internals**:

```
// Don't test that Tokio works
#[test]
fn tokio_runtime_spawns_tasks() { }
```

❌ **Rely on test execution order**:

```
// Tests should be independent
#[test]
fn test_a() { /* modifies global state */ }

#[test]
fn test_b() { /* depends on test_a */ } // ❌ Bad
```

❌ **Use magic numbers**:

```
// Bad
assert_eq!(result.len(), 42);

// Good
const EXPECTED_ITEM_COUNT: usize = 42;
assert_eq!(result.len(), EXPECTED_ITEM_COUNT);
```

Summary

Testing Strategy Highlights:

1. **Coverage Target**: 40%+ (achieved: 42-48%)
2. **Test Pyramid**: 63% unit, 33% integration, 4% E2E
3. **Critical Path Focus**: Spec-Kit (75%), MCP (65%), DB (60%)
4. **Tools**: cargo-tarpaulin, cargo-llvm-cov
5. **CI/CD**: GitHub Actions, pre-commit/pre-push hooks
6. **604 Tests Total**: 100% pass rate

Next Steps: - Test Infrastructure - MockMcpManager, fixtures - Unit Testing Guide - Patterns and examples - Integration Testing - Cross-module tests

---

References: - Rust testing guide: https://doc.rust-lang.org/book/ch11-00-testing.html - Tarpaulin docs: https://github.com/xd009642/tarpaulin - Test organization: `codex-rs/*/tests/` directories

---

Unit Testing Guide

Comprehensive guide to writing effective unit tests.

Overview

Unit Testing Philosophy: Test individual functions/components in isolation with no external dependencies

Goals: - Fast execution (<1s for all unit tests) - High coverage of business logic (70-80% for critical paths) - Deterministic and isolated - Easy to maintain

Current Status: - ~380 unit tests (63% of total) - 100% pass rate - Average execution time: ~800ms

---

Test Structure

Arrange-Act-Assert Pattern

Standard Pattern for all unit tests:

```rust
#[test]
fn test_feature_behavior() {
    // Arrange: Setup test data
    let input = "test input";
    let expected = "expected output";

    // Act: Execute function under test
    let result = function_under_test(input);

    // Assert: Verify expectations
    assert_eq!(result, expected);
}
```

Example from codebase (clarify_native.rs:365):

```rust
#[test]
fn test_vague_language_detection() {
    // Arrange
    let detector = PatternDetector::default();
    let mut issues = Vec::new();

    // Act
    detector.check_vague_language("The system should be fast", 1, &mut issues);

    // Assert
    assert_eq!(issues.len(), 1);
    assert!(issues[0].question.contains("should"));
}
```

Given-When-Then Pattern

Alternative Pattern for behavior-driven tests:

```rust
#[test]
fn test_example() {
    // Given: Initial state
    let state = StateBuilder::new("TEST").build();

    // When: Action occurs
    state.advance_stage();

    // Then: Expected outcome
    assert_eq!(state.current_stage(), Some(SpecStage::Tasks));
}
```

Naming Conventions

Test Function Names

Format: `test_{what}_{condition}_{expected}`

Good Examples:

```rust
#[test]
fn test_vague_language_detection() { }

#[test]
fn test_incomplete_markers_flagged_as_critical() { }

#[test]
fn test_quantifier_with_metrics_not_flagged() { }

#[test]
fn test_version_drift_detected_when_prd_newer() { }
```

Bad Examples:

```rust
#[test]
fn test1() { }  // ❌ Meaningless

#[test]
fn it_works() { }  // ❌ Too vague

#[test]
fn test_the_function() { }  // ❌ Not descriptive
```

Test Module Organization

In-Source Tests (preferred for unit tests):

```rust
// src/chatwidget/spec_kit/clarify_native.rs

pub fn detect_ambiguities(prd_content: &str) -> Result<Vec<AmbiguityIssue>> {
    // Implementation...
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_vague_language_detection() {
        // Test implementation...
    }

    #[test]
    fn test_incomplete_markers() {
        // Test implementation...
    }
}
```

Benefits: - ✅ Tests live next to code - ✅ Private function access - ✅ Excluded from release builds

---

Testing Pure Functions

What are Pure Functions?

Definition: Functions that: 1. Always return same output for same input 2. Have no side effects (no I/O, no mutations) 3. Don’t depend on external state

Why Test Them: Easiest to test, highest value per test

Example 1: Pattern Matching

Function (clarify_native.rs):

```
/// Check for vague language
fn check_vague_language(
    &self,
    line: &str,
    line_num: usize,
    issues: &mut Vec<AmbiguityIssue>,
) {
    for (pattern, severity, question, suggestion) in
&self.vague_patterns {
        if let Some(mat) = Regex::new(pattern).unwrap().find(line) {
            issues.push(AmbiguityIssue {
                id: format!("AMB-{:03}", issues.len() + 1),
                severity: *severity,
                pattern_name: "vague_language".to_string(),
                question: question.to_string(),
                suggestion: suggestion.to_string(),
                // ...
            });
        }
    }
}
```

Unit Test (clarify_native.rs:365):

```
#[test]
fn test_vague_language_detection() {
    let detector = PatternDetector::default();
    let mut issues = Vec::new();

    detector.check_vague_language("The system should be fast", 1,
&mut issues);

    assert_eq!(issues.len(), 1);
    assert!(issues[0].question.contains("should"));
    assert_eq!(issues[0].pattern_name, "vague_language");
}
```

What Makes This a Good Test: - ✅ Tests one specific pattern (vague language) - ✅ Verifies both detection and message content - ✅ No external dependencies - ✅ Fast (<1ms)

Example 2: Conditional Logic

Function (clarify_native.rs:385):

```
        fn check_quantifier_ambiguity(
            &self,
            line: &str,
            line_num: usize,
            issues: &mut Vec<AmbiguityIssue>,
        ) {
            for (pattern, question, suggestion) in &self.quantifier_patterns
{
                if Regex::new(pattern).unwrap().is_match(line) {
                    // Only flag if NO metrics present
                    if !has_metrics(line) {
                        issues.push(...);
                    }
                }
            }
        }
```

Unit Tests (clarify_native.rs:385):

```
        #[test]
        fn test_quantifier_ambiguity() {
            let detector = PatternDetector::default();
            let mut issues = Vec::new();

            // Should flag: no metrics
            detector.check_quantifier_ambiguity("Must be fast", 1, &mut
issues);
            assert_eq!(issues.len(), 1);

            // Should NOT flag: has metrics
            issues.clear();
            detector.check_quantifier_ambiguity("Must be fast (<100ms)", 1,
&mut issues);
            assert_eq!(issues.len(), 0);
        }
```

What Makes This a Good Test: - ✅ Tests both branches (with/without metrics) - ✅ Clear positive and negative cases - ✅ Reuses same detector (efficient)

Testing Error Handling

Testing Error Cases

Pattern: Verify function returns `Err` with expected error type

Example 1: Missing File:

```
        #[test]
```

```rust
fn test_analyze_fails_when_prd_missing() {
    let temp_dir = TempDir::new().unwrap();
    let result = check_consistency("SPEC-TEST", temp_dir.path());

    assert!(result.is_err());
    let err = result.unwrap_err();
    assert!(err.to_string().contains("PRD.md not found"));
}
```

Testing Error Messages

Pattern: Verify error messages are helpful

Example:

```rust
#[test]
fn test_error_message_includes_spec_id() {
    let result = find_spec_directory("SPEC-INVALID");

    assert!(result.is_err());
    let err = result.unwrap_err();
    assert!(err.to_string().contains("SPEC-INVALID"));
    assert!(err.to_string().contains("not found"));
}
```

Testing Panic Conditions

Use `should_panic` for panic tests:

```rust
#[test]
#[should_panic(expected = "index out of bounds")]
fn test_invalid_index_panics() {
    let stages = vec![SpecStage::Plan];
    let _ = stages[10];  // Should panic
}
```

Prefer `Result<()>` over panics:

```rust
// Good: Returns error
fn validate_index(idx: usize) -> Result<()> {
    if idx >= 6 {
        return Err(anyhow!("Index {} out of range [0, 5]", idx));
    }
    Ok(())
}

// Bad: Panics
fn validate_index(idx: usize) {
    assert!(idx < 6, "Index out of range");
}
```

Testing with Test Data

Inline Test Data

Pattern: Small data inline in test

```
#[test]
fn test_requirement_extraction() {
    let prd_content = r#"
# PRD

## Requirements

- **R1**: User can log in
- **R2**: User can log out
    "#;

    let requirements = extract_requirements(prd_content);

    assert_eq!(requirements.len(), 2);
    assert_eq!(requirements[0].id, "R1");
    assert_eq!(requirements[1].id, "R2");
}
```

External Test Fixtures

Pattern: Large data from files (see test-infrastructure.md)

```
#[test]
fn test_with_real_prd() -> Result<()> {
    let prd_path = "tests/fixtures/prds/SPEC-DEMO-prd.md";
    let content = std::fs::read_to_string(prd_path)?;

    let ambiguities = detect_ambiguities(&content)?;

    // Real PRD should have known ambiguities
    assert!(ambiguities.len() > 0);
    assert!(ambiguities.iter().any(|a| a.severity ==
Severity::Critical));

    Ok(())
}
```

Generated Test Data

Pattern: Use proptest for fuzz testing (see property-testing-guide.md)

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_regex_escape_never_panics(s in ".*") {
        // Should handle any string
        let escaped = regex_escape(&s);
        assert!(escaped.len() >= s.len());
    }
}
```

Testing State Machines

Example: SpecAutoState Transitions

State Machine: - Plan → Tasks → Implement → Validate → Audit → Unlock

Test Pattern: Verify transitions

```rust
#[test]
fn test_stage_advancement() {
    let mut state = StateBuilder::new("SPEC-TEST")
        .starting_at(SpecStage::Plan)
        .build();

    // Initial state
    assert_eq!(state.current_stage(), Some(SpecStage::Plan));
    assert_eq!(state.current_index, 0);

    // Advance to Tasks
    state.advance_stage();
    assert_eq!(state.current_stage(), Some(SpecStage::Tasks));
    assert_eq!(state.current_index, 1);

    // Advance to Implement
    state.advance_stage();
    assert_eq!(state.current_stage(), Some(SpecStage::Implement));
    assert_eq!(state.current_index, 2);
}
```

Testing Invalid Transitions

```rust
#[test]
fn test_cannot_advance_past_unlock() {
    let mut state = StateBuilder::new("SPEC-TEST")
        .starting_at(SpecStage::Unlock)
        .build();

    state.current_index = 5;  // Unlock (last stage)

    // Advancing should be no-op or return None
    state.advance_stage();
    assert_eq!(state.current_stage(), None);
}
```

Testing State Invariants

```rust
#[test]
fn test_state_index_never_negative() {
    let state = StateBuilder::new("SPEC-TEST").build();

    // Type system prevents negative (usize)
    assert!(state.current_index >= 0);

    // But ensure index is valid
    assert!(state.current_index < 6);
}
```

Testing Calculations

Scoring Functions

Example (checklist_native.rs:350):

```
        fn score_testability(prd_content: &str, issues: &mut
Vec<QualityIssue>) -> f32 {
            let mut score = 0.0;

            // Check for acceptance criteria (40%)
            let ac_re = Regex::new(r"(?mi)^###?\s+Acceptance
Criteria").unwrap();
            if ac_re.is_match(prd_content) {
                score += 40.0;
            }

            // Check for test scenarios (20%)
            let test_re = Regex::new(r"(?mi)^##\s+Test
(Strategy|Scenarios)").unwrap();
            if test_re.is_match(prd_content) {
                score += 20.0;
            }

            score.max(0.0)
        }
```

Unit Tests:

```
        #[test]
        fn test_score_testability_perfect() {
            let prd = r#"
        ### Acceptance Criteria
        - AC1: Test

        ## Test Strategy
        - Test
            "#;

            let mut issues = Vec::new();
            let score = score_testability(prd, &mut issues);

            assert_eq!(score, 60.0);  // 40 + 20
            assert_eq!(issues.len(), 0);
        }

        #[test]
        fn test_score_testability_missing_tests() {
            let prd = r#"
        ### Acceptance Criteria
        - AC1: Test
            "#;

            let mut issues = Vec::new();
            let score = score_testability(prd, &mut issues);

            assert_eq!(score, 40.0);  // 40 (AC) + 0 (no tests)
            assert!(issues.iter().any(|i| i.category == "testability"));
        }
```

```
#[test]
fn test_score_testability_zero() {
    let prd = "# PRD\n\nNo structure";

    let mut issues = Vec::new();
    let score = score_testability(prd, &mut issues);

    assert_eq!(score, 0.0);
    assert!(issues.len() > 0);
}
```

Penalty Calculations

Example (checklist_native.rs:408):

```
fn score_consistency(issues: &[InconsistencyIssue]) -> f32 {
    let critical_count = issues.iter()
        .filter(|i| matches!(i.severity, Severity::Critical))
        .count();
    let important_count = issues.iter()
        .filter(|i| matches!(i.severity, Severity::Important))
        .count();

    let penalty = (critical_count as f32 * 20.0)
                + (important_count as f32 * 10.0);

    (100.0 - penalty).max(0.0)
}
```

Unit Tests:

```
#[test]
fn test_score_consistency_perfect() {
    let issues = vec![];
    let score = score_consistency(&issues);
    assert_eq!(score, 100.0);
}

#[test]
fn test_score_consistency_one_critical() {
    let issues = vec![
        InconsistencyIssue {
            severity: Severity::Critical,
            // ...
        }
    ];
    let score = score_consistency(&issues);
    assert_eq!(score, 80.0);  // 100 - 20
}

#[test]
fn test_score_consistency_multiple_issues() {
    let issues = vec![
        InconsistencyIssue { severity: Severity::Critical, /* ... */
},
        InconsistencyIssue { severity: Severity::Critical, /* ... */
},
        InconsistencyIssue { severity: Severity::Important, /* ...
*/ },
    ];
    let score = score_consistency(&issues);
    assert_eq!(score, 50.0);  // 100 - (2*20 + 1*10)
}

#[test]
fn test_score_consistency_floor_at_zero() {
    let issues = vec![
        InconsistencyIssue { severity: Severity::Critical, /* ... */
}; 10
    ];
    let score = score_consistency(&issues);
    assert_eq!(score, 0.0);  // Floor at 0 (would be -100)
}
```

Testing Collections

Testing Filters

```
        #[test]
        fn test_filter_critical_issues() {
            let issues = vec![
                AmbiguityIssue { severity: Severity::Critical, /* ... */ },
                AmbiguityIssue { severity: Severity::Important, /* ... */ },
                AmbiguityIssue { severity: Severity::Minor, /* ... */ },
            ];

            let critical: Vec<_> = issues.iter()
                .filter(|i| matches!(i.severity, Severity::Critical))
                .collect();

            assert_eq!(critical.len(), 1);
        }
```

Testing Sorting

Example (clarify_native.rs:313):

```
        fn sort_by_severity(issues: &mut Vec<AmbiguityIssue>) {
            issues.sort_by(|a, b| match (&a.severity, &b.severity) {
                (Severity::Critical, Severity::Critical) => Ordering::Equal,
                (Severity::Critical, _) => Ordering::Less,
                (_, Severity::Critical) => Ordering::Greater,
                // ...
            });
        }
```

Unit Test:

```
        #[test]
        fn test_sort_by_severity() {
            let mut issues = vec![
                AmbiguityIssue { severity: Severity::Minor, id: "1".into(),
/* ... */ },
                AmbiguityIssue { severity: Severity::Critical, id:
"2".into(), /* ... */ },
                AmbiguityIssue { severity: Severity::Important, id: "3".into(), /* ... */ },
            ];

            sort_by_severity(&mut issues);

            assert_eq!(issues[0].severity, Severity::Critical);
            assert_eq!(issues[1].severity, Severity::Important);
            assert_eq!(issues[2].severity, Severity::Minor);
        }
```

Testing Aggregations

```
#[test]
fn test_count_by_severity() {
    let issues = vec![
        AmbiguityIssue { severity: Severity::Critical, /* ... */ },
        AmbiguityIssue { severity: Severity::Critical, /* ... */ },
        AmbiguityIssue { severity: Severity::Important, /* ... */ },
    ];

    let counts = count_by_severity(&issues);

    assert_eq!(counts.critical, 2);
    assert_eq!(counts.important, 1);
    assert_eq!(counts.minor, 0);
}
```

Testing String Manipulation

Regex Matching

```
#[test]
fn test_requirement_id_extraction() {
    let line = "- **R42**: User can authenticate";
    let re = Regex::new(r"\*\*R(\d+)\*\*").unwrap();

    let cap = re.captures(line).unwrap();
    let id = &cap[1];

    assert_eq!(id, "42");
}
```

String Transformations

Example (clarify_native.rs:334):

```
fn truncate_context(text: &str, max_len: usize) -> String {
    if text.len() <= max_len {
        text.to_string()
    } else {
        format!("{}...", &text[..max_len])
    }
}
```

Unit Tests:

```
#[test]
fn test_truncate_short_text() {
    let text = "Short";
    let result = truncate_context(text, 10);
    assert_eq!(result, "Short");
}

#[test]
fn test_truncate_long_text() {
    let text = "This is a very long text that should be truncated";
    let result = truncate_context(text, 10);
    assert_eq!(result, "This is a ...");
    assert_eq!(result.len(), 13);  // 10 + "..."
}

#[test]
fn test_truncate_exact_length() {
    let text = "Exactly10!";  // 10 chars
    let result = truncate_context(text, 10);
    assert_eq!(result, "Exactly10!");
}
```

Regex Escaping

Function (clarify_native.rs:349):

```
fn regex_escape(s: &str) -> String {
    s.chars()
        .map(|c| match c {
            '\\' | '.' | '+' | '*' | '?' | '(' | ')' | '|'
            | '[' | ']' | '{' | '}' | '^' | '$' => {
                format!("\\{}", c)
            }
            _ => c.to_string(),
        })
        .collect()
}
```

Unit Tests:

```
#[test]
fn test_regex_escape_special_chars() {
    assert_eq!(regex_escape("a.b"), "a\\.b");
    assert_eq!(regex_escape("a*b"), "a\\*b");
    assert_eq!(regex_escape("a?b"), "a\\?b");
    assert_eq!(regex_escape("a(b)"), "a\\(b\\)");
}

#[test]
fn test_regex_escape_normal_chars() {
    assert_eq!(regex_escape("abc"), "abc");
    assert_eq!(regex_escape("123"), "123");
}

#[test]
fn test_regex_escape_multiple_special() {
    assert_eq!(regex_escape("a.b*c?"), "a\\.b\\*c\\?");
}
```

Testing File Operations (with TempDir)

Setup Pattern

```
use tempfile::TempDir;

#[test]
fn test_write_and_read_prd() -> Result<()> {
    // Arrange: Create temp directory
    let temp_dir = TempDir::new()?;
    let spec_dir = temp_dir.path().join("docs/SPEC-TEST-test");
    std::fs::create_dir_all(&spec_dir)?;

    let prd_path = spec_dir.join("PRD.md");
    let content = "# PRD\n\n## Goal\nTest";

    // Act: Write file
    std::fs::write(&prd_path, content)?;

    // Assert: Read and verify
    let read_content = std::fs::read_to_string(&prd_path)?;
    assert_eq!(read_content, content);

    Ok(())
    // TempDir auto-cleaned on drop
}
```

Testing Directory Creation

```
#[test]
fn test_create_spec_directory() -> Result<()> {
    let temp_dir = TempDir::new()?;
    let spec_id = "SPEC-TEST-001";

    let spec_dir = create_spec_directory(temp_dir.path(), spec_id)?;

    assert!(spec_dir.exists());
    assert!(spec_dir.is_dir());
    assert!(spec_dir.ends_with("SPEC-TEST-001-test"));

    Ok(())
}
```

Testing with Mocks

MockMcpManager Usage

Pattern: Replace real MCP with mock

```
#[tokio::test]
async fn test_consensus_fetch() -> Result<()> {
    // Arrange: Setup mock
    let mut mock = MockMcpManager::new();
    mock.add_fixture(
        "local-memory",
        "search",
        Some("SPEC-TEST plan"),
        json!({"memory": {"content": "Agent response"}})
    );

    // Act: Call function that uses MCP
    let results = fetch_consensus("SPEC-TEST", SpecStage::Plan,
&mock).await?;

    // Assert: Verify results
    assert_eq!(results.len(), 1);

    Ok(())
}
```

See test-infrastructure.md for details.

Table-Driven Tests

Pattern: Multiple Test Cases

```
#[test]
fn test_stage_index_mapping() {
    let test_cases = vec![
        (0, Some(SpecStage::Plan)),
        (1, Some(SpecStage::Tasks)),
        (2, Some(SpecStage::Implement)),
        (3, Some(SpecStage::Validate)),
        (4, Some(SpecStage::Audit)),
        (5, Some(SpecStage::Unlock)),
        (6, None),
        (100, None),
    ];

    for (index, expected) in test_cases {
        let mut state = StateBuilder::new("SPEC-TEST").build();
        state.current_index = index;

        assert_eq!(
            state.current_stage(),
            expected,
            "Failed for index {}",
            index
        );
    }
}
```

Benefits: - ✅ Compact (many cases in one test) - ✅ Easy to add new cases - ✅ Clear failure messages

---

Parameterized Tests (with rstest)

Add to `Cargo.toml`:

```
[dev-dependencies]
rstest = "0.18"
```

Usage:

```
use rstest::rstest;

#[rstest]
#[case("should", Severity::Important)]
#[case("must", Severity::Critical)]
#[case("TBD", Severity::Critical)]
#[case("TODO", Severity::Important)]
fn test_vague_language_severity(#[case] pattern: &str, #[case]
expected: Severity) {
    let detector = PatternDetector::default();
    let mut issues = Vec::new();

    detector.check_vague_language(
        &format!("The system {} work", pattern),
        1,
        &mut issues
    );

    assert_eq!(issues.len(), 1);
    assert_eq!(issues[0].severity, expected);
}
```

Common Assertions

Equality

```
assert_eq!(actual, expected);
assert_ne!(actual, unexpected);
```

Boolean

```
assert!(condition);
assert!(!condition);
```

Contains

```
assert!(vec.contains(&item));
assert!(string.contains("substring"));
```

Custom Messages

```
assert_eq!(
    actual,
    expected,
    "Expected {}, got {} (context: {})",
    expected,
    actual,
    context
);
```

Floating Point

```
// Don't use assert_eq! for floats
// Use approx crate instead

use approx::assert_relative_eq;

assert_relative_eq!(actual, expected, epsilon = 0.001);
```

Best Practices

DO

✅ **Test one thing per test**:

```
#[test]
fn test_vague_language_detection() {
    // Only tests vague language, nothing else
}

#[test]
fn test_incomplete_markers() {
    // Only tests incomplete markers
}
```

✅ **Use descriptive names**:

```
#[test]
fn test_quantifier_with_metrics_not_flagged() {
    // Clear what's being tested
}
```

✅ **Test edge cases**:

```
#[test]
fn test_truncate_empty_string() {
    assert_eq!(truncate_context("", 10), "");
}

#[test]
fn test_score_consistency_floor_at_zero() {
    // Test penalty doesn't go negative
}
```

✅ **Keep tests independent**:

```
#[test]
fn test_a() {
    let state = StateBuilder::new("TEST-A").build();
    // Uses own state, doesn't affect other tests
}
```

```
#[test]
fn test_b() {
    let state = StateBuilder::new("TEST-B").build();
    // Independent
}
```

✅ **Use setup functions for common data**:

```
fn create_test_prd() -> String {
    r#"
# PRD
## Requirements
- **R1**: Test
    "#.to_string()
}

#[test]
fn test_with_prd() {
    let prd = create_test_prd();
    // Use prd...
}
```

DON’T

❌ **Test implementation details**:

```
// Bad: Tests internal regex pattern
#[test]
fn test_regex_pattern_is_correct() {
    assert_eq!(VAGUE_PATTERN, r"(should|could|might)");
}

// Good: Tests behavior
#[test]
fn test_vague_language_detected() {
    // Tests that "should" is flagged
}
```

❌ **Rely on test execution order**:

```
// Bad: test_b depends on test_a running first
static mut SHARED_STATE: i32 = 0;

#[test]
fn test_a() {
    unsafe { SHARED_STATE = 42; }
}

#[test]
fn test_b() {
    unsafe { assert_eq!(SHARED_STATE, 42); }  // ❌ Flaky
}
```

❌ **Use magic numbers**:

```
// Bad
```

```
assert_eq!(score, 42.0);

// Good
const EXPECTED_SCORE: f32 = 42.0;
assert_eq!(score, EXPECTED_SCORE);

// Or explain inline
assert_eq!(score, 60.0);  // 40 (AC) + 20 (test strategy)
```

✗ Test too much in one test:

```
// Bad: Tests everything at once
#[test]
fn test_entire_quality_system() {
    // 100 lines of setup
    // Tests clarify, analyze, checklist
    // Hard to debug when fails
}

// Good: Split into focused tests
#[test]
fn test_clarify_detects_vague_language() { }

#[test]
fn test_analyze_finds_missing_requirements() { }

#[test]
fn test_checklist_scores_completeness() { }
```

✗ Skip cleanup (use TempDir):

```
// Bad: Leaves files behind
#[test]
fn test_write_file() {
    std::fs::write("/tmp/test.txt", "data")?;
    // File persists after test
}

// Good: Auto-cleanup
#[test]
fn test_write_file() -> Result<()> {
    let temp_dir = TempDir::new()?;
    std::fs::write(temp_dir.path().join("test.txt"), "data")?;
    Ok(())
    // temp_dir dropped, files deleted
}
```

Running Tests

Run All Unit Tests

```
cd codex-rs
cargo test --lib
```

Explanation: - `--lib`: Only library tests (no integration tests) - Runs all `#[cfg(test)] mod tests { }` blocks

---

Run Specific Module

```
cargo test -p codex-tui --lib clarify_native
```

Breakdown: - `-p codex-tui`: Package - `--lib`: Unit tests only - `clarify_native`: Module filter

---

Run Specific Test

```
cargo test -p codex-tui test_vague_language_detection
```

Output:

```
running 1 test
test
chatwidget::spec_kit::clarify_native::tests::test_vague_language_dete
 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Run with Output

```
cargo test -- --nocapture
```

Shows `println!()` output even for passing tests.

Run with Threads

```
# Single-threaded (for debugging)
cargo test -- --test-threads=1

# Parallel (default)
cargo test -- --test-threads=8
```

Test Coverage

Measure Coverage

Using tarpaulin:

```
cargo tarpaulin -p codex-tui --lib
```

Output:

```
|| Tested/Total Lines:
|| tui/src/chatwidget/spec_kit/clarify_native.rs: 89/120
||
|| Coverage: 74.2%
```

Improve Coverage

Identify Untested Lines:

```
cargo tarpaulin -p codex-tui --lib --out Html
open target/tarpaulin/index.html
```

HTML Report shows: - ✅ Green: Covered - ❌ Red: Not covered - ⚠️ Yellow: Partially covered

---

Summary

Unit Testing Best Practices:

1. **Structure**: Use Arrange-Act-Assert pattern
2. **Naming**: `test_{what}_{condition}_{expected}`
3. **Scope**: One thing per test
4. **Independence**: No shared state
5. **Speed**: Fast (<1ms typical)
6. **Coverage**: 70-80% for critical paths
7. **Cleanup**: Use `TempDir` for filesystem tests
8. **Mocks**: Use `MockMcpManager` for MCP

Test Types Covered: - ✅ Pure functions (pattern matching, calculations) - ✅ Error handling (missing files, invalid input) - ✅ State machines (transitions, invariants) - ✅ Collections (filtering, sorting, aggregation) - ✅ String manipulation (regex, truncation, escaping) - ✅ File operations (with TempDir)

Next Steps: - Integration Testing Guide - Cross-module tests - Property Testing Guide - Generative testing - Test Infrastructure - MockMcpManager, fixtures

---

References: - Rust testing guide: https://doc.rust-lang.org/book/ch11-00-testing.html - Example tests: `codex-rs/tui/src/chatwidget/spec_kit/*/tests.rs` - Test infrastructure: `codex-rs/tui/tests/common/`
