Specification-Driven Development Guide for AI Coding Agents

A Comprehensive Framework for Documentation-First Development Workflows

Version: 1.0
Date: January 2026
Target Agent: Claude Code / AI Development Assistants

Table of Contents

1. Executive Summary

This guide establishes a specification-driven development workflow that ensures all code is preceded by comprehensive documentation. The framework is designed to be enforced automatically through configuration files, git hooks, and AI agent instructions.

> **CRITICAL PRINCIPLE: No code may be written without first creating approved specifications. This is enforced at multiple levels: agent instructions, git hooks, and CI/CD pipelines.**

Key Benefits

- Prevents undocumented code from entering the codebase
- Ensures specifications stay synchronized with implementation
- Creates a single source of truth for feature requirements
- Enables effective handoffs between developers and AI agents
- Provides comprehensive test coverage through BDD scenarios

Required Directory Structure

```
project/
├── .clinerules                    # Claude Code configuration
├── CLAUDE_CODE_INSTRUCTIONS.md    # Agent-specific instructions
├── docs/
│   ├── specs/
│   │   └── {feature-name}/
│   │       ├── README.md          # Functional spec
│   │       ├── TECHNICAL.md       # Technical design
│   │       ├── TESTING.md         # Test plan
│   │       └── scenarios.feature  # BDD scenarios
│   ├── adrs/
│   │   └── ADR-{num}-{title}.md   # Architecture decisions
│   └── templates/
│       ├── functional-spec.md
│       ├── technical-spec.md
│       ├── bdd-scenarios.feature
│       ├── test-plan.md
│       └── adr-template.md
├── scripts/
│   ├── new-feature.sh             # Feature scaffolding
│   └── check-spec-coverage.py     # Coverage monitoring
├── .git/hooks/
│   ├── pre-commit                 # Spec validation
│   └── commit-msg                 # Message validation
└── src/
    └── {feature-name}/            # Implementation
```

2. Core Configuration Files

2.1 .clinerules - Project-Level Configuration

```
.clinerules
```

```yaml
# Claude Code Project Configuration
project_name: "YourProject"
documentation_root: "docs/specs"

# CRITICAL: Enforce documentation-first workflow
workflow_phases:
  - name: "specification"
    required_before: ["implementation", "testing"]
    artifacts:
      - "functional_spec"
      - "technical_spec"
      - "bdd_scenarios"
      - "adr" # if architectural decision needed

  - name: "implementation"
    required_before: ["code_review"]
    requires_spec_approval: true

  - name: "testing"
    required_artifacts:
      - "unit_tests"
      - "integration_tests"

# Template paths
templates:
  functional_spec: "docs/templates/functional-spec.md"
  technical_spec: "docs/templates/technical-spec.md"
  bdd_scenarios: "docs/templates/bdd-scenarios.feature"
  adr: "docs/templates/adr-template.md"
  test_plan: "docs/templates/test-plan.md"

# Automatic documentation rules
documentation_rules:
  - trigger: "code_change"
    action: "check_spec_sync"
    description: "Verify specs reflect actual implementation"

  - trigger: "bug_fix"
    action: "update_edge_cases"
    description: "Add bug scenario to BDD specs as regression test"

  - trigger: "api_change"
    action: "update_openapi"
    description: "Regenerate API documentation"

# File naming conventions
naming_conventions:
  specs: "docs/specs/{feature-name}/README.md"
  adrs: "docs/adrs/ADR-{number}-{title}.md"
  bdd: "tests/features/{feature-name}.feature"
  test_plans: "docs/specs/{feature-name}/testing.md"
```

```
# Code patterns to enforce
code_standards:
  - "Use repository pattern for data access"
  - "Follow existing logging patterns from utils/logger.py"
  - "All API endpoints must have OpenAPI annotations"
  - "All services must have corresponding unit tests"

# Pre-implementation checklist
required_before_coding:
  - "Functional specification approved"
  - "Technical design reviewed"
  - "BDD scenarios written"
  - "Edge cases documented"
  - "Success metrics defined"
```

2.2 CLAUDE_CODE_INSTRUCTIONS.md - Agent Instructions

CLAUDE_CODE_INSTRUCTIONS.md

```
# Claude Code Development Instructions

## CRITICAL: Documentation-First Workflow

Before writing ANY code, you MUST create comprehensive specifications.
This is NON-NEGOTIABLE and follows this exact order:

### Phase 1: Specification (ALWAYS START HERE)

When I request a new feature, your FIRST response must be:

"I'll create the specification documents first. This includes:
1. Functional Specification
2. Technical Design
3. BDD Scenarios
4. Architecture Decision Record (if applicable)
5. Test Plan

I'll use the templates in docs/templates/ and create these in docs/specs/{feature-name}/

Let me start with the functional specification..."

Then create ALL specification documents before asking permission to implement.

### Required Specification Structure

For each new feature, create:

docs/specs/{feature-name}/
├── README.md              # Functional spec (user-facing)
├── TECHNICAL.md           # Technical design
├── TESTING.md             # Test plan
└── scenarios.feature      # BDD scenarios

If architectural decision needed:
docs/adrs/ADR-{next-number}-{title}.md

### Specification Templates

[TEMPLATES DETAILED IN SECTION 3 BELOW]

### Phase 2: Specification Review

After creating all specs, you MUST:
1. Present summary of created specifications
2. Ask: "Please review these specifications. Once approved, I'll proceed with implementat
3. WAIT for explicit approval before coding

### Phase 3: Implementation

Only after spec approval:
1. Create git branch: feature/{feature-name}
```

````
2. Implement following specs exactly
3. Write tests matching BDD scenarios
4. Update specs if implementation reveals issues

### Phase 4: Documentation Sync

After ANY code change:
1. Check if specs need updates
2. Update relevant sections
3. Add "Last Updated" timestamp
4. If bug fix: Add scenario to BDD specs
5. If API change: Update OpenAPI/technical docs

## Documentation Update Triggers

### When fixing a bug:

Action: Add to scenarios.feature
```gherkin
Scenario: Bug #123 - [description]
  # Regression test for bug fix
  Given [condition that caused bug]
  When [action]
  Then [correct behavior]
```

Update README.md edge cases section with the bug scenario

### When modifying APIs:
1. Update TECHNICAL.md API section
2. Update OpenAPI spec (if exists)
3. Update example requests/responses

### When changing data models:
1. Update TECHNICAL.md data model section
2. Document migration steps
3. Update example data in tests

## File Organization Rules

Always maintain this structure:
project/
├── docs/
│   ├── specs/
│   │   └── {feature-name}/
│   │       ├── README.md         # Functional
│   │       ├── TECHNICAL.md      # Technical
│   │       ├── TESTING.md        # Test plan
│   │       └── scenarios.feature # BDD
│   ├── adrs/
│   │   └── ADR-{num}-{title}.md
│   └── templates/
│       └── [template files]
````

```
├── tests/
│   ├── features/
│   │   └── {feature-name}.feature  # Copy of BDD scenarios for pytest-bdd
│   ├── unit/
│   └── integration/
└── src/
    └── [implementation]

## Naming Conventions

- Feature branches: feature/{feature-name}
- Spec folders: docs/specs/{feature-name}
- ADR files: ADR-{number}-{kebab-case-title}.md
- BDD files: {feature-name}.feature

## Example Workflow

User: "I need to add email notification when calendar sync fails"

You respond:
"I'll create the specification documents first for the email notification feature.

Creating docs/specs/email-sync-notifications/:
1. README.md - Functional specification
2. TECHNICAL.md - Technical design
3. TESTING.md - Test plan
4. scenarios.feature - BDD scenarios

Let me start..."

[Then create all four documents]

[Then present summary and wait for approval]

[Only then start coding]

## NEVER Skip These Steps

❌ Don't start with: "I'll implement..."
✅ Always start with: "I'll create the specification first..."

❌ Don't skip specs for "small changes"
✅ Every change gets documented proportionally

❌ Don't forget to update specs after code changes
✅ Always check spec sync after implementation

## Verification Checklist

Before marking any task complete:
- [ ] All specification documents created
- [ ] BDD scenarios cover happy path + edge cases
- [ ] Technical design documents implementation approach
```

```
- [ ] Test plan defines verification strategy
- [ ] Code matches specifications
- [ ] Specs updated if implementation deviated
- [ ] All tests passing
- [ ] Documentation up to date
```

3. Specification Templates

3.1 Functional Specification Template

docs/templates/functional-spec.md

```
# [Feature Name]

**Status**: Draft | In Review | Approved | Implemented
**Owner**: [Name]
**Created**: YYYY-MM-DD
**Last Updated**: YYYY-MM-DD

## Objective
[One sentence: what and why]

## User Stories

### Story 1: [Title]
**As a** [role]
**I want** [feature]
**So that** [benefit]

**Acceptance Criteria**:
- [ ] Criterion 1
- [ ] Criterion 2
- [ ] Criterion 3

### Story 2: [Title]
**As a** [role]
**I want** [feature]
**So that** [benefit]

**Acceptance Criteria**:
- [ ] Criterion 1
- [ ] Criterion 2

## Success Metrics
- Metric 1: [how to measure - must be quantifiable]
- Metric 2: [how to measure - must be quantifiable]
- Metric 3: [how to measure - must be quantifiable]

## User Flow
1. User action 1
2. System response 1
3. User action 2
4. System response 2
5. Final outcome

## Edge Cases
1. Case 1: [specific scenario] → Expected behavior
2. Case 2: [specific scenario] → Expected behavior
3. Case 3: [specific scenario] → Expected behavior

## Error Scenarios
1. Error 1: [condition] → User sees [message] → System [action]
2. Error 2: [condition] → User sees [message] → System [action]
```

```
## Dependencies
- Dependency 1: [description]
- Dependency 2: [description]

## Out of Scope
- Item 1
- Item 2
- Item 3

## Future Enhancements
- Enhancement 1: [description]
- Enhancement 2: [description]
```

3.2 Technical Specification Template

docs/templates/technical-spec.md

````
# Technical Design: [Feature Name]

**Last Updated**: YYYY-MM-DD

## Architecture Overview
[High-level design description]
[Include diagrams if needed - Mermaid or PlantUML preferred]

## Components

### New Components
- **Component Name**: Purpose and responsibilities
  - Location: `src/path/to/component.py`
  - Responsibilities:
    - Responsibility 1
    - Responsibility 2
  - Dependencies:
    - Dependency 1
    - Dependency 2
  - Interfaces:
    - Interface 1: description
    - Interface 2: description

### Modified Components
- **Component Name**: Changes needed
  - Current behavior: [describe]
  - New behavior: [describe]
  - Impact: [describe potential impacts]

## Data Model Changes

### New Tables/Collections
```sql
CREATE TABLE example (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  name VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX idx_name (name)
);
```

### Schema Modifications
```sql
ALTER TABLE existing_table
ADD COLUMN new_field VARCHAR(100);

CREATE INDEX idx_new_field ON existing_table(new_field);
```

### Schema Migrations
- Migration 001: [description of change]
````

````
  - Up: [SQL or commands to apply]
  - Down: [SQL or commands to rollback]

## API Design

### New Endpoints

#### POST /api/v1/resource
**Purpose**: [What this endpoint does]

**Request**:
```json
{
  "field1": "string",
  "field2": 123,
  "field3": {
    "nested": "object"
  }
}
```

**Response** (200 OK):
```json
{
  "id": "uuid",
  "status": "success",
  "data": {
    "field1": "value"
  }
}
```

**Errors**:
- 400 Bad Request: Invalid input
  ```json
  {
    "error": "validation_error",
    "details": ["field1 is required"]
  }
  ```
- 401 Unauthorized: Invalid credentials
- 500 Internal Server Error: Server error

**Rate Limiting**: 100 requests/minute per user

### Modified Endpoints
- **PUT /api/v1/resource/{id}**
  - Changes: [describe modifications]
  - Breaking: Yes/No
  - Migration path: [if breaking]

## Implementation Details
````

````
### Algorithm/Logic
```python
# Pseudocode or detailed description
def process_sync(source, target):
    # Step 1: Validate inputs
    validate_credentials(source, target)

    # Step 2: Fetch data
    data = fetch_from_source(source)

    # Step 3: Transform
    transformed = transform_data(data)

    # Step 4: Sync to target
    result = sync_to_target(target, transformed)

    # Step 5: Log and return
    log_sync_event(result)
    return result
```

### Error Handling Strategy
- **Network Errors**: Retry with exponential backoff (max 3 attempts)
- **Validation Errors**: Return 400 with detailed error messages
- **Authentication Errors**: Return 401, log security event
- **Server Errors**: Return 500, alert on-call engineer

### Performance Considerations
- Expected load: [requests/sec, data volume]
- Optimization strategy:
  - Caching: [what and how]
  - Indexing: [database indexes]
  - Async processing: [what operations]
  - Connection pooling: [configuration]

### Concurrency Handling
- Locking strategy: [optimistic/pessimistic]
- Race condition prevention: [approach]
- Idempotency: [how ensured]

## Security Considerations

### Authentication
- Method: OAuth2 / JWT / API Key
- Token expiration: [duration]
- Refresh strategy: [approach]

### Authorization
- Role-based access: [roles and permissions]
- Resource-level permissions: [approach]

### Data Protection
- Encryption at rest: [method]
````

````
- Encryption in transit: TLS 1.3
- PII handling: [approach]
- Sensitive data: [logging restrictions]

### Input Validation
- Sanitization: [approach]
- XSS prevention: [method]
- SQL injection prevention: Parameterized queries

## Dependencies

### External Services
- **Service Name**: [purpose]
  - Endpoint: https://api.example.com
  - Authentication: [method]
  - Rate limits: [limits]
  - Fallback: [strategy if unavailable]

### Libraries
- **library-name@version**: [purpose]
  - License: [type]
  - Security concerns: [any]

### Internal Services
- **Service Name**: [dependency description]
  - Contract: [API version]
  - Failure mode: [how to handle]

## Configuration

### Required Config Changes
```yaml
# config.yaml
feature_name:
  enabled: true
  option1: value1
  option2: value2

# Environment variables
FEATURE_API_KEY: "secret"
FEATURE_ENDPOINT: "https://api.example.com"
```

### Feature Flags
- `feature.email-notifications.enabled`: Enable email notifications
- `feature.email-notifications.rate-limit`: Max emails per hour

## Monitoring & Observability

### Metrics to Track
- `email.sent.count`: Counter - Total emails sent
- `email.failed.count`: Counter - Failed email attempts
- `email.send.duration`: Histogram - Time to send email
````

```
- `email.queue.size`: Gauge - Current queue size

### Logs to Emit
```python
logger.info("Email sent", extra={
    "user_id": user.id,
    "email_type": "notification",
    "template": "sync-failed",
    "duration_ms": 125
})
```

Alerts to Configure
- Email send failure rate > 5% for 5 minutes
- Email queue size > 1000 for 10 minutes
- Email send duration p95 > 5 seconds

Dashboards
- Email service health dashboard
- Per-user email metrics
- Template performance metrics

Testing Strategy

Unit Tests
- Test coverage target: 90%
- Key areas:
  - Input validation
  - Business logic
  - Error handling

Integration Tests
- External service mocking strategy
- Database transaction handling
- End-to-end flows

Performance Tests
- Load test scenarios: [describe]
- Expected performance: [metrics]
- Stress test limits: [thresholds]

Rollout Plan

Phase 1: Internal Testing (Week 1)
- Deploy to development environment
- Internal team testing
- Metrics validation

Phase 2: Beta Release (Week 2)
- Enable for 10% of users
- Monitor metrics closely
- Gather feedback
```

```
### Phase 3: General Availability (Week 3)
- Enable for all users
- Continue monitoring
- Announce feature

## Rollback Strategy

### Rollback Triggers
- Error rate > 5%
- Performance degradation > 20%
- Critical bug discovered

### Rollback Process
1. Disable feature flag
2. Revert database migrations (if safe)
3. Notify stakeholders
4. Investigate root cause

### Data Handling
- In-flight requests: [how handled]
- Partial data: [cleanup strategy]

## Open Questions
- [ ] Question 1: [needs clarification from]
- [ ] Question 2: [needs decision on]
```

3.3 BDD Scenarios Template

docs/templates/bdd-scenarios.feature

```
Feature: [Feature Name]
  As a [role]
  I want [feature]
  So that [benefit]

  Background:
    Given the system is configured with [setup details]
    And the user is authenticated as [role]
    And the database contains [initial state]

  Scenario: Happy path - [primary use case description]
    Given [specific precondition 1]
    And [specific precondition 2]
    When [user action or event]
    Then [expected outcome 1]
    And [expected outcome 2]
    And the system logs [expected log entry]

  Scenario: Edge case - [specific edge case name]
    Given [edge case precondition]
    And [edge case setup]
    When [action that triggers edge case]
    Then [expected edge case handling]
    And [system state verification]

  Scenario Outline: Multiple valid inputs - [variation description]
    Given a user with role ""
    When they perform action with "
                                   "
    Then they should see ""
    And the system records ""

    Examples:
      | role    | input     | output      | metric        |
      | admin   | value1    | result1     | admin_action  |
      | user    | value2    | result2     | user_action   |
      | guest   | value3    | result3     | guest_action  |

  Scenario: Error case - [specific error scenario]
    Given [precondition for error]
    When [action that causes error]
    Then the user should see error message "[exact message]"
    And the system should log error with code "[error_code]"
    And the system should NOT [undesired side effect]

  Scenario: Performance requirement - [performance scenario]
    Given [performance test setup]
    And [load conditions]
    When [action under load]
    Then response time should be less than [X] milliseconds
    And all requests should succeed
```

```
  Scenario: Security requirement - [security scenario]
    Given an unauthenticated user
    When they attempt to [protected action]
    Then they should receive 401 Unauthorized
    And the attempt should be logged

  Scenario: Idempotency - [idempotency scenario]
    Given [initial state]
    When the same request is made [N] times
    Then the result should be identical
    And only [N] records should exist in the database

  # Regression tests for bugs
  Scenario: Bug #XXX - [bug description]
    # Regression test added: YYYY-MM-DD
    Given [condition that caused the bug]
    When [action that triggered the bug]
    Then [correct behavior that was missing]
    And [verification that bug is fixed]
```

3.4 Test Plan Template

docs/templates/test-plan.md

```
# Test Plan: [Feature Name]

**Last Updated**: YYYY-MM-DD

## Test Strategy

### Testing Pyramid
- **Unit Tests**: 70% coverage - Fast, isolated tests
- **Integration Tests**: 20% coverage - Component interaction
- **End-to-End Tests**: 10% coverage - Full user flows

### Test Environments
- **Development**: Local development testing
- **Staging**: Pre-production validation
- **Production**: Canary and smoke tests

## Unit Tests

### Component 1: [Component Name]
**Location**: `tests/unit/test_component1.py`

#### Test Cases
- **test_valid_input**: Verifies correct processing of valid input
  - Input: [example]
  - Expected: [result]
  - Coverage: Happy path

- **test_invalid_input**: Verifies error handling
  - Input: [invalid example]
  - Expected: [error type]
  - Coverage: Validation logic

- **test_edge_case**: Verifies boundary conditions
  - Input: [edge case]
  - Expected: [handling]
  - Coverage: Edge cases

**Coverage Target**: 95%

### Component 2: [Component Name]
**Location**: `tests/unit/test_component2.py`

[Similar structure as above]

## Integration Tests

### Flow 1: [Integration Flow Name]
**Location**: `tests/integration/test_flow1.py`

**Purpose**: Verify interaction between [Component A] and [Component B]

**Setup Requirements**:
```

```
- Database with test schema
- Mock external API responses
- Test user credentials

**Test Scenario**:
```python
def test_complete_flow():
    # Arrange
    setup_test_data()

    # Act
    result = perform_integration_action()

    # Assert
    assert result.status == "success"
    assert database_state_correct()
    assert external_api_called_correctly()
```

Assertions:
- Database state matches expected
- External services called with correct parameters
- Response format is correct
- Error handling works across components

Flow 2: [Integration Flow Name]
[Similar structure]

End-to-End Tests

Scenario 1: [E2E Scenario Name]
Location: `tests/e2e/test_scenario1.py`

Purpose: Verify complete user workflow from [start] to [end]

Preconditions:
- Clean test environment
- Test user account exists
- External services in known state

Test Steps:
1. User logs in with credentials
2. User navigates to [page]
3. User performs [action]
4. System displays [result]
5. User verifies [outcome]

Verification Points:
- UI displays correct information
- Database updated correctly
- Email sent (if applicable)
- Audit log contains entry
```

````
**Cleanup**:
- Remove test data
- Reset system state

## Test Data

### Test Users
```python
test_users = [
    {
        "email": "admin@test.com",
        "role": "admin",
        "permissions": ["read", "write", "delete"]
    },
    {
        "email": "user@test.com",
        "role": "user",
        "permissions": ["read", "write"]
    }
]
```

### Test Data Sets
```python
valid_inputs = [
    {"field1": "value1", "field2": 123},
    {"field1": "value2", "field2": 456},
]

invalid_inputs = [
    {"field1": "", "field2": -1},  # Empty and negative
    {"field1": "x" * 1000, "field2": 0},  # Too long
]

edge_cases = [
    {"field1": "min", "field2": 1},  # Minimum values
    {"field1": "max", "field2": 999999},  # Maximum values
]
```

## Performance Tests

### Load Test 1: [Scenario Name]
**Tool**: Locust / JMeter / k6

**Scenario**: [Description]

**Configuration**:
- Users: 100 concurrent
- Duration: 10 minutes
- Ramp-up: 30 seconds

**Expected Results**:
````

```
- Average response time: < 500ms
- 95th percentile: < 1000ms
- 99th percentile: < 2000ms
- Error rate: < 0.1%

**Metrics to Collect**:
- Requests per second
- Response time distribution
- Error rate by endpoint
- Database query times

## Security Tests

### Penetration Testing
- [ ] SQL injection attempts
- [ ] XSS attack vectors
- [ ] Authentication bypass attempts
- [ ] Authorization checks
- [ ] Rate limiting validation

### Security Scanning
- [ ] OWASP ZAP scan
- [ ] Dependency vulnerability scan
- [ ] Secret detection scan

## Manual Testing Checklist

### Functional Testing
- [ ] All user stories tested
- [ ] All acceptance criteria met
- [ ] Edge cases verified
- [ ] Error messages user-friendly

### Usability Testing
- [ ] UI/UX flows intuitive
- [ ] Responsive on mobile
- [ ] Accessible (WCAG 2.1 AA)
- [ ] Loading states clear

### Cross-Browser Testing
- [ ] Chrome (latest)
- [ ] Firefox (latest)
- [ ] Safari (latest)
- [ ] Edge (latest)

### Cross-Device Testing
- [ ] Desktop (1920x1080)
- [ ] Tablet (iPad)
- [ ] Mobile (iPhone, Android)

## Regression Tests

### Critical Paths to Verify
```

```
- [ ] User authentication flow
- [ ] Core feature X still works
- [ ] Integration with service Y intact
- [ ] Performance hasn't degraded

### Automated Regression Suite
**Location**: `tests/regression/`
**Frequency**: Run on every commit
**Duration**: < 15 minutes

## Test Execution Schedule

### Per Commit (CI Pipeline)
- Unit tests
- Linting
- Type checking
- Security scanning

### Per Pull Request
- Integration tests
- E2E tests (smoke suite)
- Code coverage report

### Nightly
- Full E2E suite
- Performance tests
- Security scans

### Pre-Release
- Full regression suite
- Load testing
- Manual exploratory testing
- Cross-browser testing

## Test Reporting

### Metrics to Track
- Test coverage percentage
- Test execution time
- Pass/fail rate
- Flaky test count

### Reports Generated
- JUnit XML for CI integration
- HTML coverage reports
- Performance test results
- Security scan findings

## Risks and Mitigations

### Risk 1: External Service Unavailable
**Mitigation**: Use mocking in tests, verify mock accuracy
```

```
### Risk 2: Test Data Pollution
**Mitigation**: Isolated test databases, cleanup after each test

### Risk 3: Flaky Tests
**Mitigation**: Retry mechanism, proper wait conditions, investigation of failures
```

3.5 Architecture Decision Record Template

docs/templates/adr-template.md

```
# ADR-XXX: [Decision Title]

**Date**: YYYY-MM-DD
**Status**: Proposed | Accepted | Deprecated | Superseded
**Deciders**: [Names of decision makers]
**Technical Story**: [Ticket/Issue reference]

## Context and Problem Statement

[Describe the context and problem statement, e.g., in free form using two to three senten

## Decision Drivers

* [driver 1, e.g., a force, facing concern, ...]
* [driver 2, e.g., a force, facing concern, ...]
* [driver 3, e.g., a force, facing concern, ...]

## Considered Options

* [option 1]
* [option 2]
* [option 3]

## Decision Outcome

Chosen option: "[option X]", because [justification. e.g., only option that meets criteri

### Positive Consequences

* [e.g., improvement of quality attribute satisfaction, follow-up decisions required, ...
* [...]

### Negative Consequences

* [e.g., compromising quality attribute, follow-up decisions required, ...]
* [...]

## Pros and Cons of the Options

### [option 1]

[example | description | pointer to more information | ...]

**Pros**:
* Good, because [argument a]
* Good, because [argument b]

**Cons**:
* Bad, because [argument c]
* Bad, because [argument d]

### [option 2]
```

```
[example | description | pointer to more information | ...]

**Pros**:
* Good, because [argument a]
* Good, because [argument b]

**Cons**:
* Bad, because [argument c]
* Bad, because [argument d]

## Links

* [Link type] [Link to ADR]
* [Link type] [Link to ADR]

## Implementation Notes

[Any specific implementation guidance or constraints resulting from this decision]

## Review Schedule

[When should this decision be reviewed? e.g., After 6 months, When technology X matures]
```

4. Automation Scripts

4.1 Feature Scaffolding Script

scripts/new-feature.sh

```
#!/bin/bash
# Script to scaffold new feature with all documentation

set -e

FEATURE_NAME=$1

if [ -z "$FEATURE_NAME" ]; then
    echo "Usage: ./scripts/new-feature.sh <feature-name>"
    exit 1
fi

SPEC_DIR="docs/specs/$FEATURE_NAME"
TODAY=$(date +%Y-%m-%d)

# Create directory structure
mkdir -p "$SPEC_DIR"
mkdir -p "tests/features"
mkdir -p "src/$FEATURE_NAME"

echo "📝 Creating specification documents for: $FEATURE_NAME"

# Create functional spec from template
cat > "$SPEC_DIR/README.md" <<EOF
# $FEATURE_NAME

**Status**: Draft
**Owner**: TBD
**Created**: $TODAY
**Last Updated**: $TODAY

## Objective
[One sentence: what and why]

## User Stories

### Story 1: [Title]
**As a** [role]
**I want** [feature]
**So that** [benefit]

**Acceptance Criteria**:
- [ ] Criterion 1
- [ ] Criterion 2

## Success Metrics
- Metric 1: [how to measure]

## User Flow
1. Step 1
2. Step 2
```

```
## Edge Cases
1. Case 1: [scenario] → Expected behavior

## Out of Scope
- Item 1
EOF

# Create technical spec
cat > "$SPEC_DIR/TECHNICAL.md" <<EOF
# Technical Design: $FEATURE_NAME

**Last Updated**: $TODAY

## Architecture Overview
[High-level design]

## Components

### New Components
- **Component**: Purpose

## Implementation Details
[Details]

## Dependencies
[List]

## Monitoring
[Metrics and logs]
EOF

# Create test plan
cat > "$SPEC_DIR/TESTING.md" <<EOF
# Test Plan: $FEATURE_NAME

**Last Updated**: $TODAY

## Test Strategy

### Unit Tests
- Test 1: [description]

### Integration Tests
- Test 1: [description]

## Manual Testing Checklist
- [ ] Test scenario 1
EOF

# Create BDD scenarios
cat > "$SPEC_DIR/scenarios.feature" <<EOF
Feature: $FEATURE_NAME
  As a [role]
```

```
  I want [feature]
  So that [benefit]

  Scenario: Happy path
    Given [precondition]
    When [action]
    Then [expected outcome]
EOF

# Copy BDD to tests
cp "$SPEC_DIR/scenarios.feature" "tests/features/$FEATURE_NAME.feature"

# Create feature branch
git checkout -b "feature/$FEATURE_NAME"

echo "✅ Created specification structure:"
echo "   $SPEC_DIR/README.md"
echo "   $SPEC_DIR/TECHNICAL.md"
echo "   $SPEC_DIR/TESTING.md"
echo "   $SPEC_DIR/scenarios.feature"
echo ""
echo "📝 Next steps:"
echo "   1. Fill in specification details"
echo "   2. Get specs reviewed and approved"
echo "   3. Begin implementation"
echo ""
echo "Branch created: feature/$FEATURE_NAME"
```

> **Make executable:** `chmod +x scripts/new-feature.sh`

4.2 Specification Coverage Monitoring

scripts/check-spec-coverage.py

```
#!/usr/bin/env python3
"""Check specification coverage for all source code."""

import os
from pathlib import Path
from datetime import datetime, timedelta
import sys

def check_spec_coverage():
    """Check that all source features have specifications."""
    src_dirs = set()
    spec_dirs = set()

    # Find all source directories
    for root, dirs, files in os.walk('src'):
        # Skip __pycache__ and hidden directories
        dirs[:] = [d for d in dirs if not d.startswith('.') and d != '__pycache__']

        if files:  # Has files
            parts = Path(root).parts
            if len(parts) > 1:
                feature = parts[1]
                src_dirs.add(feature)

    # Find all spec directories
    if os.path.exists('docs/specs'):
        for item in os.listdir('docs/specs'):
            spec_path = os.path.join('docs/specs', item)
            if os.path.isdir(spec_path):
                spec_dirs.add(item)

    # Calculate coverage
    missing_specs = src_dirs - spec_dirs
    orphaned_specs = spec_dirs - src_dirs

    coverage = (len(spec_dirs) / len(src_dirs) * 100) if src_dirs else 100

    print("📊 Specification Coverage Report")
    print("=" * 60)
    print(f"Source features: {len(src_dirs)}")
    print(f"Documented features: {len(spec_dirs)}")
    print(f"Coverage: {coverage:.1f}%")
    print()

    exit_code = 0

    if missing_specs:
        print(f"❌ Missing specs for: {', '.join(sorted(missing_specs))}")
        exit_code = 1
    else:
        print(f"✅ All features have specifications")
```

```
    if orphaned_specs:
        print(f"⚠️  Orphaned specs (no code): {', '.join(sorted(orphaned_specs))}")

    # Check for outdated specs
    print()
    print("📅 Specification Freshness Check")
    print("=" * 60)

    outdated = []
    for spec_dir in sorted(spec_dirs):
        readme = f"docs/specs/{spec_dir}/README.md"
        if os.path.exists(readme):
            with open(readme) as f:
                content = f.read()
                if 'Last Updated:' in content:
                    import re
                    match = re.search(r'Last Updated:\s*(\d{4}-\d{2}-\d{2})', content)
                    if match:
                        last_updated = datetime.strptime(match.group(1), '%Y-%m-%d')
                        age_days = (datetime.now() - last_updated).days
                        if age_days > 30:
                            outdated.append((spec_dir, age_days))

    if outdated:
        print("⏰ Specs not updated in 30+ days:")
        for spec, age in sorted(outdated, key=lambda x: x[1], reverse=True):
            print(f"   - {spec}: {age} days old")
    else:
        print("✅ All specifications are up to date")

    # Check for required files in each spec
    print()
    print("📋 Required Documentation Check")
    print("=" * 60)

    required_files = ['README.md', 'TECHNICAL.md', 'TESTING.md', 'scenarios.feature']
    incomplete_specs = []

    for spec_dir in sorted(spec_dirs):
        missing_files = []
        for req_file in required_files:
            file_path = f"docs/specs/{spec_dir}/{req_file}"
            if not os.path.exists(file_path):
                missing_files.append(req_file)

        if missing_files:
            incomplete_specs.append((spec_dir, missing_files))

    if incomplete_specs:
        print("❌ Incomplete specifications:")
        for spec, missing in incomplete_specs:
            print(f"   - {spec}: missing {', '.join(missing)}")
        exit_code = 1
```

```
    else:
        print("✅ All specifications have required documentation")

    print()
    return exit_code

if __name__ == "__main__":
    sys.exit(check_spec_coverage())
```

> **Make executable:** `chmod +x scripts/check-spec-coverage.py`

4.3 Git Hooks

Pre-Commit Hook

.git/hooks/pre-commit

```
#!/bin/bash
# Pre-commit hook to enforce documentation standards

echo "🔍 Checking documentation compliance..."

# Function to check if specs exist for changed files
check_specs() {
    local changed_files=$(git diff --cached --name-only --diff-filter=ACM)
    local missing_specs=()
    local features_checked=()

    for file in $changed_files; do
        # Skip if file is in docs/ or tests/ or config files
        if [[ $file == docs/* ]] || [[ $file == tests/* ]] || \
           [[ $file == .* ]] || [[ $file == *.md ]] || \
           [[ $file == scripts/* ]]; then
            continue
        fi

        # Extract feature name from path
        if [[ $file == src/* ]]; then
            feature_name=$(echo $file | cut -d'/' -f2)

            # Skip if already checked
            if [[ " ${features_checked[@]} " =~ " ${feature_name} " ]]; then
                continue
            fi
            features_checked+=("$feature_name")

            spec_dir="docs/specs/$feature_name"

            # Check if spec directory exists
            if [[ ! -d "$spec_dir" ]]; then
                missing_specs+=("$feature_name")
            else
                # Check if all required docs exist
                required_docs=("README.md" "TECHNICAL.md" "TESTING.md" "scenarios.feature
                for doc in "${required_docs[@]}"; do
                    if [[ ! -f "$spec_dir/$doc" ]]; then
                        echo "❌ Missing $doc for $feature_name"
                        echo "   Create it with: ./scripts/new-feature.sh $feature_name"
                        exit 1
                    fi
                done
            fi
        fi
    done

    if [ ${#missing_specs[@]} -gt 0 ]; then
        echo "❌ Missing specifications for features: ${missing_specs[*]}"
        echo ""
        echo "Create specs with: ./scripts/new-feature.sh <feature-name>"
```

```
        echo ""
        echo "Required structure:"
        echo "  docs/specs/{feature-name}/"
        echo "    ├── README.md         # Functional spec"
        echo "    ├── TECHNICAL.md      # Technical design"
        echo "    ├── TESTING.md        # Test plan"
        echo "    └── scenarios.feature # BDD scenarios"
        exit 1
    fi
}

# Check if this is a feature branch
current_branch=$(git symbolic-ref --short HEAD 2>/dev/null || echo "detached")
if [[ $current_branch == feature/* ]]; then
    check_specs
fi

# Check for outdated "Last Updated" timestamps
check_timestamps() {
    local spec_files=$(find docs/specs -name "README.md" -o -name "TECHNICAL.md" -o -name
    local today=$(date +%Y-%m-%d)
    local warnings=0

    for spec in $spec_files; do
        if grep -q "Last Updated:" "$spec" 2>/dev/null; then
            last_updated=$(grep "Last Updated:" "$spec" | head -1 | sed 's/.*Last Updated
            if [[ "$last_updated" != "$today" ]]; then
                # Check if related code was modified
                feature_name=$(echo $spec | cut -d'/' -f3)
                if git diff --cached --name-only | grep -q "src/$feature_name" 2>/dev/nul
                    echo "⚠️  $spec needs timestamp update (code changed today)"
                    echo "   Update 'Last Updated: $today'"
                    warnings=$((warnings + 1))
                fi
            fi
        fi
    done

    if [ $warnings -gt 0 ]; then
        echo ""
        echo "ℹ️  Please update the 'Last Updated' timestamps in the specs above"
    fi
}

check_timestamps

echo "✅ Documentation compliance check passed"
```

Commit Message Hook

```
.git/hooks/commit-msg
```

```
#!/bin/bash
# Enforce commit message format and spec references

commit_msg_file=$1
commit_msg=$(cat "$commit_msg_file")

# Check if commit follows conventional commits format
if [[ $commit_msg =~ ^(feat|fix|docs|refactor|test|chore|style|perf|ci|build): ]]; then
    # Extract type and scope
    commit_type=$(echo "$commit_msg" | cut -d':' -f1 | cut -d'(' -f1)

    # For feature commits, check if spec exists
    if [[ $commit_type == "feat" ]]; then
        # Try to extract feature name from message or branch
        if [[ $commit_msg =~ feat\(([a-z-]+)\): ]]; then
            feature_name="${BASH_REMATCH[1]}"
        else
            # Try to get from branch name
            branch=$(git symbolic-ref --short HEAD 2>/dev/null)
            if [[ $branch =~ feature/([a-z-]+) ]]; then
                feature_name="${BASH_REMATCH[1]}"
            fi
        fi

        if [[ -n "$feature_name" ]]; then
            spec_dir="docs/specs/$feature_name"
            if [[ ! -d "$spec_dir" ]]; then
                echo "❌ No specification found for feature: $feature_name"
                echo "   Create with: ./scripts/new-feature.sh $feature_name"
                exit 1
            fi
        fi
    fi

    # For bug fixes, remind to add BDD scenario
    if [[ $commit_type == "fix" ]]; then
        echo ""
        echo "✅ Bug fix commit detected"
        echo "⚠️  Reminder: Did you add this bug scenario to BDD specs for regression tes
        echo ""
    fi
else
    echo "❌ Commit message must follow Conventional Commits format:"
    echo ""
    echo "Format: <type>(<scope>): <description>"
    echo ""
    echo "Types:"
    echo "  feat:     New feature"
    echo "  fix:      Bug fix"
    echo "  docs:     Documentation changes"
    echo "  refactor: Code refactoring"
    echo "  test:     Test additions/changes"
```

```
    echo "  chore:    Maintenance tasks"
    echo "  style:    Code style changes"
    echo "  perf:     Performance improvements"
    echo ""
    echo "Examples:"
    echo "  feat(email-sync): Add notification on sync failure"
    echo "  fix(calendar): Handle timezone edge case"
    echo "  docs(api): Update endpoint documentation"
    echo ""
    exit 1
fi
```

Make executable:

`chmod +x .git/hooks/pre-commit`

`chmod +x .git/hooks/commit-msg`

5. Multi-Agent Configuration

For projects using multiple Claude Code agents with git worktrees, create agent-specific configurations:

Spec Writer Agent

`.claude/agents/spec-writer/.clinerules`

```
agent_role: "Specification Writer"
agent_id: "spec-writer"
primary_responsibility: "Create and maintain specifications"

allowed_operations:
  - create_specs
  - update_specs
  - review_specs
  - scaffold_structure
  - update_documentation

restricted_operations:
  - write_implementation_code
  - modify_tests
  - deploy_code

auto_actions:
  on_new_feature_request:
    - run: "./scripts/new-feature.sh {feature_name}"
    - create: "all specification documents"
    - present: "summary for review"
    - wait: "approval before notifying implementation agent"

  on_code_change_detected:
    - check: "spec synchronization"
    - flag: "outdated specifications"
    - update: "Last Updated timestamps"
    - notify: "user of discrepancies"

workflow:
  1. Receive feature request
  2. Create all specification documents
  3. Fill in with comprehensive details
  4. Present to user for review
  5. Wait for approval
  6. Hand off to implementation agent

communication:
  with_implementation_agent:
    - "Specifications approved for {feature_name}"
    - "Specs updated, please sync implementation"

  with_user:
    - Always ask for approval before implementation
    - Highlight areas needing clarification

quality_checks:
  - All required docs created
  - BDD scenarios cover edge cases
  - Success metrics are quantifiable
  - Technical design is implementable
```

Implementation Agent

`.claude/agents/implementation/.clinerules`

```
agent_role: "Implementation"
agent_id: "implementation"
primary_responsibility: "Write code according to approved specs"

prerequisites:
  - spec_status: "approved"
  - specs_exist: true
  - all_required_docs_present: true

allowed_operations:
  - write_code
  - create_tests
  - refactor_code
  - fix_bugs

restricted_operations:
  - modify_specs_without_consultation
  - skip_tests
  - deploy_without_approval

auto_actions:
  before_coding:
    - verify: "spec approval status"
    - load: "technical design from TECHNICAL.md"
    - load: "BDD scenarios from scenarios.feature"
    - load: "test plan from TESTING.md"
    - create: "feature branch"

  during_coding:
    - follow: "technical design exactly"
    - implement: "tests matching BDD scenarios"
    - check: "code standards compliance"

  after_coding:
    - run: "all tests"
    - check: "spec deviation"
    - update: "specs if deviated (with approval)"
    - notify: "spec-writer agent if updates needed"

workflow:
  1. Wait for spec approval notification
  2. Review all specification documents
  3. Create implementation plan
  4. Implement feature following specs
  5. Write tests matching BDD scenarios
  6. Verify all acceptance criteria met
  7. Check for spec deviations
  8. Update specs if needed (with approval)

communication:
  with_spec_writer_agent:
    - "Implementation deviated from spec in {area}, please review"
```

```
    - "Spec needs update for {reason}"

  with_test_agent:
    - "Implementation complete, ready for testing"
    - "Following test plan in TESTING.md"

quality_checks:
  - Code matches technical design
  - All BDD scenarios pass
  - No linting errors
  - Type checking passes
  - All tests green
  - Documentation updated
```

Test Agent

.claude/agents/test/.clinerules

```
agent_role: "Test Engineer"
agent_id: "test-engineer"
primary_responsibility: "Verify implementation against specifications"

prerequisites:
  - implementation_complete: true
  - test_plan_exists: true
  - bdd_scenarios_defined: true

allowed_operations:
  - write_tests
  - run_tests
  - analyze_coverage
  - report_defects

restricted_operations:
  - modify_implementation_code
  - change_specifications
  - skip_test_scenarios

auto_actions:
  on_implementation_complete:
    - load: "test plan from TESTING.md"
    - load: "BDD scenarios from scenarios.feature"
    - verify: "all scenarios implemented as tests"
    - run: "full test suite"
    - check: "coverage requirements met"

  on_bug_found:
    - create: "bug report"
    - add: "BDD scenario for regression"
    - notify: "implementation agent"
    - update: "edge cases in functional spec"

workflow:
  1. Review test plan and BDD scenarios
  2. Verify all scenarios have tests
  3. Run comprehensive test suite
  4. Analyze coverage and gaps
  5. Report results and defects
  6. Ensure regression scenarios added

communication:
  with_implementation_agent:
    - "Tests failing: {details}"
    - "Coverage gap in {area}"

  with_spec_writer_agent:
    - "Found edge case not in specs: {case}"
    - "Suggest adding scenario: {scenario}"

quality_checks:
```

```
  - All BDD scenarios tested
  - Coverage > 90%
  - No flaky tests
  - Performance requirements met
  - Security tests pass
```

6. CI/CD Integration

GitHub Actions Workflow

`.github/workflows/spec-enforcement.yml`

```yaml
name: Specification Enforcement

on:
  push:
    branches: [ main, develop, 'feature/**' ]
  pull_request:
    branches: [ main, develop ]

jobs:
  check-specifications:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'

      - name: Check specification coverage
        run: |
          python scripts/check-spec-coverage.py

      - name: Verify required documentation
        run: |
          EXIT_CODE=0
          for dir in docs/specs/*/; do
            if [ -d "$dir" ]; then
              feature=$(basename "$dir")
              echo "Checking $feature..."

              for doc in README.md TECHNICAL.md TESTING.md scenarios.feature; do
                if [ ! -f "$dir$doc" ]; then
                  echo "❌ Missing $doc in $feature"
                  EXIT_CODE=1
                fi
              done

              # Check for Last Updated timestamp
              if ! grep -q "Last Updated:" "$dir/README.md"; then
                echo "⚠️  Missing 'Last Updated' in $feature/README.md"
              fi
            fi
          done
          exit $EXIT_CODE

      - name: Validate BDD scenarios syntax
        run: |
          # Install gherkin linter if needed
          pip install gherkin-linter
```

```
        # Validate all .feature files
        find tests/features -name "*.feature" -exec gherkin-linter {} \;

    - name: Check for orphaned code
      run: |
        # Find source directories without specs
        for src_dir in src/*/; do
          if [ -d "$src_dir" ]; then
            feature=$(basename "$src_dir")
            if [ ! -d "docs/specs/$feature" ]; then
              echo "❌ Code exists without spec: $feature"
              exit 1
            fi
          fi
        done

    - name: Generate coverage report
      run: |
        python scripts/check-spec-coverage.py > spec-coverage-report.txt
        cat spec-coverage-report.txt

    - name: Upload coverage report
      uses: actions/upload-artifact@v3
      with:
        name: spec-coverage-report
        path: spec-coverage-report.txt

  lint-and-test:
    runs-on: ubuntu-latest
    needs: check-specifications

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest pytest-cov pytest-bdd

      - name: Run linting
        run: |
          pip install ruff mypy
          ruff check .
          mypy src/

      - name: Run tests
```

```
        run: |
          pytest tests/ --cov=src --cov-report=xml --cov-report=html

      - name: Check coverage threshold
        run: |
          coverage report --fail-under=90

  validate-pr:
    runs-on: ubuntu-latest
    if: github.event_name == 'pull_request'

    steps:
      - name: Checkout code
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      - name: Check PR has spec updates
        run: |
          # Get changed files in PR
          CHANGED_FILES=$(git diff --name-only origin/${{ github.base_ref }}...HEAD)

          # Check if code changed
          if echo "$CHANGED_FILES" | grep -q "^src/"; then
            # Check if specs also changed
            if ! echo "$CHANGED_FILES" | grep -q "^docs/specs/"; then
              echo "❌ Code changes detected without spec updates"
              echo "Please update relevant specifications"
              exit 1
            fi
          fi

      - name: Validate commit messages
        run: |
          # Check all commits in PR follow conventional commits
          git log origin/${{ github.base_ref }}..HEAD --format=%s | while read msg; do
            if ! echo "$msg" | grep -qE "^(feat|fix|docs|refactor|test|chore|style|perf|c
              echo "❌ Invalid commit message: $msg"
              echo "Must follow: type(scope): description"
              exit 1
            fi
          done
```

7. n8n Workflow Automation

> This workflow monitors GitHub events and enforces specification requirements. Import into your n8n instance.

n8n-workflow-spec-enforcement.json

```
{
  "name": "Specification Enforcement Workflow",
  "nodes": [
    {
      "parameters": {
        "events": ["push", "pull_request"],
        "repository": "owner/repo"
      },
      "name": "GitHub Webhook",
      "type": "n8n-nodes-base.githubTrigger",
      "position": [250, 300]
    },
    {
      "parameters": {
        "functionCode": "// Extract event type and details\nconst eventType = $input.item
      },
      "name": "Parse Event",
      "type": "n8n-nodes-base.function",
      "position": [450, 300]
    },
    {
      "parameters": {
        "conditions": {
          "boolean": [{
            "value1": "={{$json.eventType}}",
            "value2": "push"
          }]
        }
      },
      "name": "Is Push Event?",
      "type": "n8n-nodes-base.if",
      "position": [650, 300]
    },
    {
      "parameters": {
        "functionCode": "// Check if code files were changed\nconst changedFiles = $input
      },
      "name": "Check Code Changes",
      "type": "n8n-nodes-base.function",
      "position": [850, 200]
    },
    {
      "parameters": {
        "functionCode": "// For each feature, construct expected spec paths\nconst featur
      },
      "name": "Build Spec Paths",
      "type": "n8n-nodes-base.function",
      "position": [1050, 200]
    },
    {
      "parameters": {
        "resource": "file",
```

```
        "operation": "get",
        "owner": "={{$json.repository.split('/')[0]}}",
        "repository": "={{$json.repository.split('/')[1]}}",
        "filePath": "={{$json.specsToCheck[0].paths[0]}}"
      },
      "name": "Check Spec Exists",
      "type": "n8n-nodes-base.github",
      "position": [1250, 200]
    },
    {
      "parameters": {
        "conditions": {
          "boolean": [{
            "value1": "={{$json.content}}",
            "operation": "notEqual",
            "value2": null
          }]
        }
      },
      "name": "Spec Exists?",
      "type": "n8n-nodes-base.if",
      "position": [1450, 200]
    },
    {
      "parameters": {
        "resource": "issue",
        "operation": "create",
        "owner": "={{$json.repository.split('/')[0]}}",
        "repository": "={{$json.repository.split('/')[1]}}",
        "title": "Missing Specification",
        "body": "## ⚠️ Missing Specification Detected\n\nCode changes were pushed without
        "labels": ["documentation", "required", "automated"]
      },
      "name": "Create Issue",
      "type": "n8n-nodes-base.github",
      "position": [1650, 100]
    },
    {
      "parameters": {
        "functionCode": "// Check Last Updated timestamp\nconst content = Buffer.from($js
      },
      "name": "Check Timestamp",
      "type": "n8n-nodes-base.function",
      "position": [1650, 300]
    },
    {
      "parameters": {
        "conditions": {
          "boolean": [{
            "value1": "={{$json.needsUpdate}}",
            "value2": true
          }]
        }
```

```
      },
      "name": "Needs Update?",
      "type": "n8n-nodes-base.if",
      "position": [1850, 300]
    },
    {
      "parameters": {
        "resource": "issue",
        "operation": "createComment",
        "owner": "={{$json.repository.split('/')[0]}}",
        "repository": "={{$json.repository.split('/')[1]}}",
        "issueNumber": "={{$json.relatedIssue}}",
        "body": "⏰ **Specification Timestamp Update Needed**\n\nThe specification for `{
      },
      "name": "Comment on PR",
      "type": "n8n-nodes-base.github",
      "position": [2050, 400]
    }
  ],
  "connections": {
    "GitHub Webhook": {
      "main": [[{"node": "Parse Event", "type": "main", "index": 0}]]
    },
    "Parse Event": {
      "main": [[{"node": "Is Push Event?", "type": "main", "index": 0}]]
    },
    "Is Push Event?": {
      "main": [
        [{"node": "Check Code Changes", "type": "main", "index": 0}],
        []
      ]
    },
    "Check Code Changes": {
      "main": [[{"node": "Build Spec Paths", "type": "main", "index": 0}]]
    },
    "Build Spec Paths": {
      "main": [[{"node": "Check Spec Exists", "type": "main", "index": 0}]]
    },
    "Check Spec Exists": {
      "main": [[{"node": "Spec Exists?", "type": "main", "index": 0}]]
    },
    "Spec Exists?": {
      "main": [
        [{"node": "Check Timestamp", "type": "main", "index": 0}],
        [{"node": "Create Issue", "type": "main", "index": 0}]
      ]
    },
    "Check Timestamp": {
      "main": [[{"node": "Needs Update?", "type": "main", "index": 0}]]
    },
    "Needs Update?": {
      "main": [
        [{"node": "Comment on PR", "type": "main", "index": 0}],
```

```
          []
        ]
      }
    }
  }
```

8. Implementation Instructions for AI Agent

INSTRUCTIONS FOR AI AGENT READING THIS DOCUMENT: When this document is uploaded to a project repository, you should: 1. Create the directory structure as defined in Section 1 2. Create all configuration files: - .clinerules - CLAUDE_CODE_INSTRUCTIONS.md 3. Create all template files in docs/templates/ 4. Create all automation scripts: - scripts/new-feature.sh - scripts/check-spec-coverage.py 5. Create git hooks: - .git/hooks/pre-commit - .git/hooks/commit-msg 6. Make scripts executable with chmod +x 7. Create multi-agent configurations if requested 8. Create CI/CD workflow files if requested 9. Test the setup by running: - ./scripts/new-feature.sh test-feature - ./scripts/check-spec-coverage.py After setup, you should: - Inform the user that the specification-driven workflow is now active - Explain that ALL future feature requests will follow the spec-first process - Show the user how to create a new feature using the scaffolding script - Remind the user to install git hooks with: git config core.hooksPath .git/hooks

Checklist for AI Agent

- ☐ Create docs/specs/ directory
- ☐ Create docs/adrs/ directory
- ☐ Create docs/templates/ directory
- ☐ Create scripts/ directory
- ☐ Create .clinerules file
- ☐ Create CLAUDE_CODE_INSTRUCTIONS.md file
- ☐ Create all template files in docs/templates/
- ☐ Create scripts/new-feature.sh

- ☐ Create scripts/check-spec-coverage.py
- ☐ Make scripts executable
- ☐ Create pre-commit hook
- ☐ Create commit-msg hook
- ☐ Make hooks executable
- ☐ Create .claude/agents/ directory (if multi-agent)
- ☐ Create agent-specific .clinerules (if multi-agent)
- ☐ Create .github/workflows/ directory (if CI/CD requested)
- ☐ Create spec-enforcement.yml workflow (if CI/CD requested)
- ☐ Test scaffolding script
- ☐ Verify git hooks work
- ☐ Run coverage check script
- ☐ Report completion to user

Verification Steps

After creating all files, the AI agent should verify:

```
# Test feature scaffolding
./scripts/new-feature.sh test-feature

# Verify structure created
ls -la docs/specs/test-feature/

# Should show:
# - README.md
# - TECHNICAL.md
# - TESTING.md
# - scenarios.feature

# Test coverage script
./scripts/check-spec-coverage.py

# Should output specification coverage report

# Test git hooks (if git initialized)
git add .
git commit -m "test: verify commit hook"

# Should validate commit message format
```

User Onboarding Message

After setup, the AI agent should present this message to the user:

✅ Specification-Driven Development Workflow Activated

Your project is now configured for specification-first development. Here's what changed:

📁 New Directory Structure

- docs/specs/ - Feature specifications
- docs/adrs/ - Architecture decision records
- docs/templates/ - Specification templates
- scripts/ - Automation scripts

🔧 New Commands

- `./scripts/new-feature.sh <name>` - Create new feature with full specs
- `./scripts/check-spec-coverage.py` - Check specification coverage

🎯 Workflow Changes

- All feature requests now start with specification creation
- Implementation only begins after spec approval
- Git hooks enforce documentation requirements
- CI/CD validates specification compliance

🚀 Next Steps

1. Try creating a feature: `./scripts/new-feature.sh my-first-feature`
2. Fill in the specifications in docs/specs/my-first-feature/
3. Approve the specs
4. I'll implement following the approved specifications

From now on, when you request a feature, I will always create comprehensive specifications first and wait for your approval before writing any code.