Morph Management

AI AGENTS & SKILLS

Building Intelligent Development Workflows with Claude Code

A Guide to Autonomous Development Pipelines

From Requirements to Delivery · Automated Bug Resolution · Human-in-the-Loop Review

February 2026

1. Executive Summary

Software development teams face mounting pressure: faster delivery, higher quality, and shrinking teams. AI Agents — autonomous, goal-directed programs powered by large language models — offer a transformative approach. Rather than replacing developers, they act as tireless collaborators, handling repetitive tasks, accelerating feedback loops, and surfacing insights that would otherwise be buried in logs and tickets.

This paper explains what AI Agents are, how they acquire specialised capabilities through Skills, and how they can be orchestrated — using Anthropic's Claude Code as the underlying engine — into complete development workflows. Two reference workflows are explored in depth: a human-guided pipeline from requirements to delivery, and a fully automated bug-resolution loop triggered by a Jira event.

By the end of this paper you will understand:

- What an AI Agent is and how it differs from a simple chatbot or script
- How Skills give agents specialised, reusable capabilities
- How Claude Code provides the agentic runtime that makes this possible
- How to design, configure, and evolve multi-agent workflows
- How human oversight is built into the architecture — not bolted on

2. Introduction: Why Agentic AI in Development?

2.1 The Challenge

Modern software delivery is complex. A typical feature request touches requirements analysis, system design, coding, test writing, code review, CI/CD configuration, staging, and monitoring. Each step introduces delay and the potential for human error, especially when context is lost between handoffs.

Traditional automation – shell scripts, CI pipelines, bots – helps with well-defined, repetitive tasks. But they break when requirements are ambiguous, when edge cases arise, or when work requires synthesising knowledge across multiple systems. This is exactly where AI Agents excel.

2.2 What Has Changed

Three developments have converged to make agentic AI practical in 2025–26:

1. Large language models (LLMs) now possess sufficient reasoning capability to plan multi-step tasks, evaluate intermediate results, and adapt when plans fail.
2. Tool-use and function-calling APIs allow LLMs to interact with external systems – reading/writing files, calling APIs, running tests – rather than just generating text.
3. Persistent context and memory mechanisms allow agents to maintain state across extended tasks, remembering earlier decisions and outputs.

Claude Code, Anthropic's agentic coding system, brings all three together in a command-line driven solution which allows developers to interact effectively with their existing tools, while using the appropriate LLM model to add value, at all levels. With Claude Code, you can read and write code, run shell commands, call APIs, browse documentation, and coordinate with other agents – all within a principled safety framework.

The LLMs models on offer, from Anthropic, notably Opus and Sonnet, now at version 4.6 are arguably the leader in software creation, and the various processes involved, from design to coding to documentation. While various adaptations are possible to support 3rd party models, or indeed locally hosted models, the capabilities offered by Claude Code with the Anthropic models make this the top framework on the block for agentic based development.

3. Understanding AI Agents

3.1 Definition

An AI Agent is a software system that:

- Perceives its environment (reads files, consumes API responses, receives messages)
- Maintains a goal or set of objectives given by an operator or another agent
- Plans a sequence of actions to achieve those objectives
- Executes those actions using available tools
- Evaluates results and adapts its plan if outcomes are unexpected
- Reports progress, asks for clarification, or escalates when it cannot proceed

This is fundamentally different from a chatbot (which only responds conversationally) or a script (which executes a fixed sequence regardless of context). An agent is goal-driven and adaptive.

3.2 The Agent Loop

Internally, most LLM-based agents operate in a continuous loop:

The Agent Reasoning Loop (ReAct Pattern)
1. OBSERVE – Read current state: task description, prior outputs, tool results
2. THINK – Reason about what to do next; identify the best action
3. ACT – Call a tool, write a file, invoke an API, or send a message
4. OBSERVE – Receive the result of the action
5. EVALUATE – Did the action succeed? Is the goal closer? Any errors?
6. REPEAT – Continue until goal is achieved, or escalate if stuck

This loop continues until the agent's goal is satisfied, a stopping condition is met, or the agent determines it needs human input. Claude Code implements this loop natively, with configurable limits on iterations, token budgets, and escalation conditions.

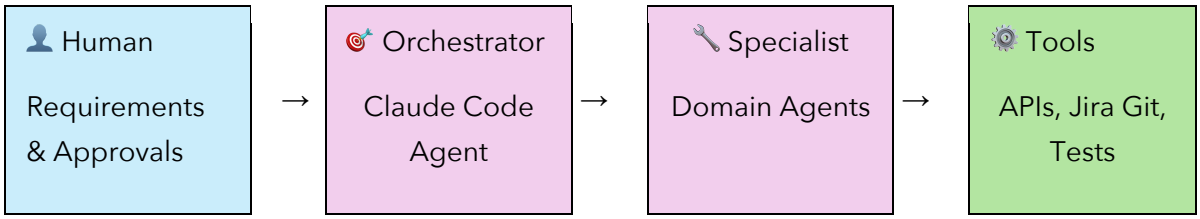
3.3 Types of Agents in a Development Workflow

Not all agents are equal. In a multi-agent development system, you typically encounter several distinct roles:

Agent Role	Primary Responsibility
Orchestrator Agent	Receives the top-level goal, decomposes it into sub-tasks, routes work to specialist agents, and aggregates results.
Requirements Agent	Analyses user stories, asks clarifying questions, and produces structured acceptance criteria.
Architecture Agent	Designs system structure, selects patterns, and documents decisions as Architecture Decision Records.
Code Agent	Writes, refactors, and documents implementation code following project standards.
Test Agent	Generates test plans, writes automated tests, and analyses coverage gaps.
Review Agent	Performs automated code review, checking style, security, and correctness.
Debug / Triage Agent	Analyses logs, traces errors to root causes, and proposes fixes.
Deployment Agent	Manages CI/CD pipeline configuration and monitors release health.

3.4 High-Level Architecture

The diagram below illustrates the four-tier architecture of a multi-agent development system, from human interaction at the top, through orchestration and specialist agents, down to the tools and external services they use:



Each tier has a distinct responsibility. Humans set goals and approve key decisions. The Orchestrator coordinates the overall plan. Specialist Agents apply domain expertise. Tools are the hands that interact with real systems – Git, Jira, test runners, cloud APIs.

3.5 Agent Communication

Agents communicate by passing structured messages – typically JSON or Markdown – through the orchestrator. A common pattern is:

```
# Example agent message (YAML format)
```

```
from: requirements_agent
to:   architecture_agent
task: design_feature
payload:
  feature_id:    FEAT-042
  user_story:    'As a user I want to export reports as PDF'
  constraints:   ['must use existing auth service', 'PDF < 5MB']
  acceptance:    ['PDF matches on-screen layout', 'export < 3s for 50-page report']
priority: HIGH
awaiting_human_approval: false
```

This example of a message an agent handling incoming requirements might send to an architecture agent request it to design the feature. The fields included are relevant to the application and purpose.

4. Skills: Giving Agents Specialised Capabilities

4.1 What is a Skill?

A Skill is a reusable, self-contained package of instructions, context, and configuration that teaches an agent how to perform a specific task well. Think of a Skill as a job description and procedure manual rolled into one – it tells the agent not just what to do, but how to do it, what inputs to expect, what outputs to produce, and what to do when things go wrong.

Skills are typically stored as Markdown files (often named SKILL.md) within a project. They are loaded into an agent's context at runtime – either always, or conditionally when the orchestrator determines a skill is needed. This keeps each agent's context lean and focused. So, when an agent starts, it gets the details of its job description from the skill.

While skills were originally specified by Anthropic, they have become the standard used by all of the AI development frameworks, such as GitHub Copilot or Cursor.

4.2 Anatomy of a Skill File

A well-structured Skill file contains the following sections:

Section	Purpose & Content
Frontmatter (YAML)	name, description, trigger conditions – tells the orchestrator when to load this skill
Objective	One-sentence statement of what this skill accomplishes

Inputs	Required context, parameters, and data the skill expects to receive
Process Steps	Numbered steps the agent follows, referencing tools and sub-skills
Outputs	Expected artefacts: files, Jira updates, messages, test results
Error Handling	What to do when steps fail – retry, escalate, or fallback actions
Configuration	Environment variables, API endpoints, thresholds that can be tuned
Examples	Sample invocations showing inputs → expected outputs

4.3 Example Skill: Code Review

Here is an illustrative example of a code-review skill definition. In practice this would be a full SKILL.md file in your repository:

```
---
name: code-review
description:
  Performs a structured review of a pull request. Check for: correctness,
  test coverage, security issues, style violations, and documentation gaps.
triggers:
  - pull_request_opened
  - pull_request_updated
---
## Objective
Produce an actionable PR review that unblocks the human reviewer.

## Inputs
- pr_diff:      The unified diff of the PR
- pr_title:     PR title and description
- coding_guide: Path to CODING_STANDARDS.md
- test_report:  Latest CI test results JSON

## Process Steps
1. Read the diff and identify files changed
2. For each file: check style against coding_guide
3. Identify functions without test coverage in test_report
4. Scan for common security patterns (SQL injection, unvalidated input)
5. Draft review comments with LINE REFERENCES
6. Summarise: APPROVE / REQUEST_CHANGES / COMMENT
```

```
## Outputs
- GitHub PR review (posted via GitHub API)
- Structured JSON: { decision, comments: [{file, line, severity, text}] }

## Configuration
- MIN_TEST_COVERAGE: 80   # fail if below this threshold
- SECURITY_SCAN: true
- POST_TO_GITHUB: true
```

The details are pretty understandable and based on the available tools, and from the inputs and outputs involved, the skill details are sufficient to get Claude to do what you want. Of course to get it working exactly as intended, it may require a little trial and error, with adjustments.

4.4 Skills vs Agent Identity

It is important to distinguish between an agent and its skills.

- **An Agent** is the runtime entity – it has memory, a conversation history, and executes the agent loop.
- **A Skill** is a behaviour specification that the agent loads and follows.

One agent can hold multiple skills. One skill can be shared across multiple agents.

This separation is powerful: you can improve the code-review skill without modifying any agent configuration, and you can deploy a new agent that reuses existing, proven skills without writing new logic from scratch.

4.5 The Agent–Skill Relationship

The table below summarises the key agents in a development workflow and the skills each one employs:

AGENT		SKILLS / CAPABILITIES
🎯 Orchestrator Agent	→	▸ Route tasks to specialist agents ▸ Manage state & task queue ▸ Handle retries & escalations ▸ Report progress to humans
📋 Requirements Agent	→	▸ Parse natural language specs ▸ Generate user stories & acceptance criteria ▸ Identify ambiguities & request clarification ▸ Create structured requirement documents

🏗 Architecture Agent	→	▸ Evaluate tech stack options ▸ Design system diagrams & data models ▸ Identify non-functional requirements ▸ Produce ADRs (Architecture Decision Records)
💻 Code Agent	→	▸ Write clean, documented code ▸ Follow project coding standards ▸ Generate unit tests alongside code ▸ Create pull requests with descriptions
🧪 Test Agent	→	▸ Design test plans & edge cases ▸ Run automated test suites ▸ Analyse coverage reports ▸ Log bugs with full reproduction steps
🔬 Debug Agent	→	▸ Parse logs & stack traces ▸ Correlate errors across services ▸ Suggest root causes with evidence ▸ Verify fixes with regression tests

5. Claude Code: The Agentic Runtime

5.1 What is Claude Code?

Claude Code is Anthropic's command-line agentic system that runs Claude in a persistent, tool-enabled environment. Unlike a simple API call that returns a text response, Claude Code:

- Maintains a persistent session with memory of prior steps
- Has direct access to the file system, terminal, and network
- Can execute shell commands, run test suites, and call APIs
- Supports a Model Context Protocol (MCP) to connect arbitrary external tools
- Can spawn sub-agents and delegate tasks to them
- Respects configurable safety boundaries (e.g. cannot delete production databases)

Claude Code is the engine that brings agents to life. When you define an agent with a set of skills and point it at Claude Code, you get an autonomous system that can plan, execute, and adapt across multi-step engineering tasks.

5.2 Key Capabilities Relevant to Development Workflows

Capability	How It Powers Agent Workflows
File System Access	Agents read source code, write new files, update configs, and commit changes to Git – all without manual intervention.
Shell Execution	Agents run test suites, linters, build tools, and deployment scripts and interpret the results.
API & Tool Calls (MCP)	Agents connect to Jira, GitHub, Slack, monitoring tools, and any service with an API – reading tickets, posting reviews, sending alerts.
Sub-agent Delegation	The orchestrator spawns specialist sub-agents for focused tasks, managing them as a team rather than doing everything itself.
Context Window Management	Claude Code manages long-running tasks by summarising earlier work and prioritising relevant context – avoiding hallucination from overloaded prompts.
Safety Controls	Operators configure exactly what Claude Code can and cannot do – e.g. 'can run tests but cannot deploy to production without human approval'.
Audit Trail	Every action Claude Code takes is logged with reasoning, making the agent's decision-making transparent and debuggable.

5.3 The CLAUDE.md Configuration File

Claude Code looks for a CLAUDE.md file at the root of your project. This file serves as the project-specific instruction set – it tells Claude about your codebase conventions, the tools available, the workflow to follow, and any constraints that apply.

```
# CLAUDE.md – Project Configuration for Claude Code

## Project Overview
This is a Python/FastAPI microservice. Always follow PEP 8 style.
Test framework: pytest. Coverage target: 85%.

## Available Skills
- skills/code-review/SKILL.md      # Use when reviewing PRs
- skills/test-generation/SKILL.md  # Use when writing tests
```

```
- skills/bug-triage/SKILL.md       # Use when investigating failures

## Workflow Rules
1. Never push directly to main — always create a feature branch
2. All changes require a passing CI pipeline before PR creation
3. Security issues (HIGH or CRITICAL) must be escalated to human
4. When coverage drops below 80%, generate missing tests before proceeding

## Tool Configuration
- jira_project: MYAPP
- github_repo:  org/myapp
- slack_channel: #dev-agents

## Human Escalation Points
- Architecture decisions that affect >3 services
- Estimated effort > 5 story points
- Any change to authentication or payment flows
```

5.4 Model Context Protocol (MCP)

MCP is an open standard developed by Anthropic that allows Claude Code to connect to any external service through a standardised interface. Each MCP server exposes a set of tools – functions the agent can call. Examples in a development workflow:

- Jira MCP Server: create_issue, update_issue, get_sprint, add_comment
- GitHub MCP Server: create_pr, get_diff, post_review, merge_pr
- Monitoring MCP Server: get_error_rate, fetch_logs, get_traces, alert_on_threshold
- CI/CD MCP Server: trigger_pipeline, get_pipeline_status, get_test_results

To connect a tool, you add a single entry to Claude Code's MCP configuration. The agent then discovers the tool automatically and can use it in any skill. The following is an example of MCP to Jira and GitHub making them available to Claude Code.

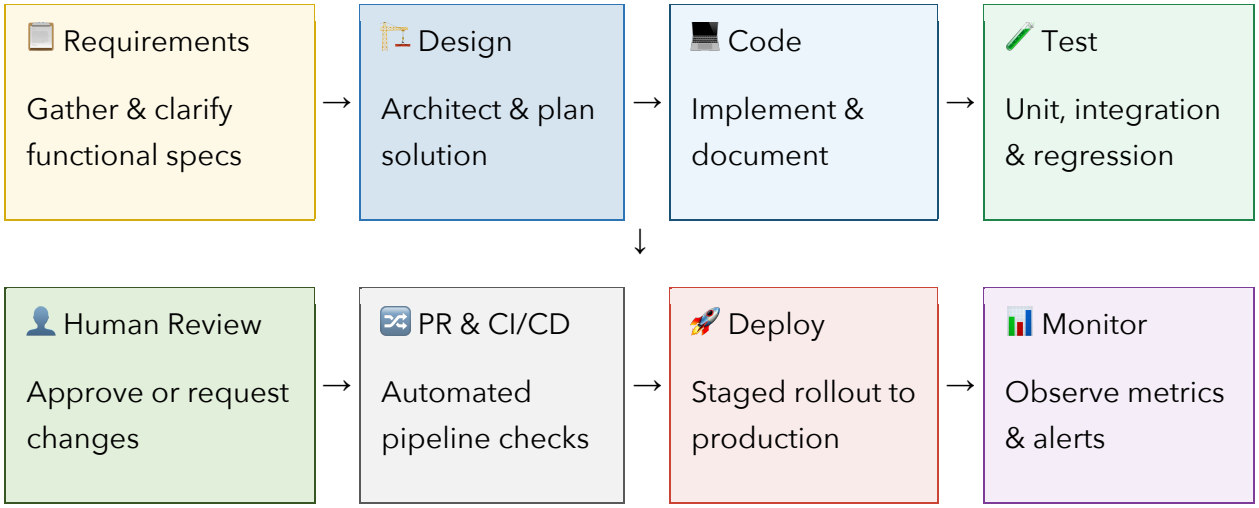
```
# .claude/mcp_settings.json
{
  'mcpServers': {
    'jira': {
      'command': 'npx',
      'args': ['@modelcontextprotocol/server-jira'],
      'env': { 'JIRA_URL': 'https://company.atlassian.net',
            'JIRA_TOKEN': '${JIRA_API_TOKEN}' }
```

```
    },
    'github': {
      'command': 'npx',
      'args': ['@modelcontextprotocol/server-github'],
      'env': { 'GITHUB_TOKEN': '${GITHUB_PAT}' }
    }
  }
}
```

6. Workflow 1: Requirements to Delivery

6.1 Overview

This workflow models a human-in-the-loop development pipeline. A human provides the initial requirement; agents handle analysis, design, implementation, and testing; the human reviews and approves before deployment. This is the most common starting pattern for teams adopting agentic AI – it builds trust incrementally by keeping humans in control of key decisions.



6.2 Stage-by-Stage Walkthrough

Stage 1: Requirements Capture

The human enters a requirement – as a free-form description, a Jira ticket, or a meeting note. The Requirements Agent loads the requirements-capture skill and:

1. Reads the input and identifies the core user need
2. Searches existing documentation for related features or constraints
3. Generates structured user stories with Given/When/Then acceptance criteria
4. Posts clarifying questions to a Slack channel (or Jira comment) if ambiguities are found
5. Awaits human confirmation before proceeding

This stage ensures that vague requirements are caught early, before any code is written. The output is a structured requirements document committed to the project repository.

Stage 2: 🏗️ Architecture & Design

Once requirements are approved, the Architecture Agent examines the existing codebase, the tech stack, and any relevant Architecture Decision Records (ADRs). It:

1. Identifies which existing components will be affected
2. Proposes a solution approach with options (with tradeoffs)
3. Creates or updates ADRs for significant decisions
4. Estimates complexity and flags if human architectural review is needed

If the change is large (e.g., introduces a new service or changes a data model), an automatic escalation to human review is triggered at this point — before design time is wasted on an approach that will be rejected.

Stage 3: 💻 Code Implementation

The Code Agent receives the approved design and:

1. Creates a feature branch from main
2. Implements the code following CLAUDE.md project standards
3. Writes inline documentation and updates any README sections affected
4. Runs the linter and type checker, fixing any issues
5. Commits changes with a meaningful commit message

Stage 4: 🧪 Test Generation & Execution

The Test Agent reviews the new code and the acceptance criteria, then:

1. Writes unit tests for all new functions and edge cases
2. Writes integration tests for any new API endpoints or database interactions
3. Runs the full test suite and checks coverage
4. If coverage is below the threshold, writes additional tests
5. Attaches a test report to the Jira ticket

Stage 5: 👤 Human Review

A pull request is created automatically, with:

- A description linking to the original requirement and ADR
- An automated code review summary from the Review Agent
- Test coverage and CI pipeline status
- A brief explanation of design choices and alternatives considered

The human developer reviews the PR. They can approve, request changes (which triggers another Code Agent iteration), or escalate for additional human review. This is the primary human control point in the pipeline.

Stage 6: 🚀 Deployment & Monitoring

Once approved, the Deployment Agent:

1. Merges the PR to the staging branch and monitors the CI pipeline
2. Runs smoke tests in the staging environment
3. If staging is healthy, promotes to production using a canary deployment
4. Monitors error rates and latency for 15 minutes post-deployment
5. Rolls back automatically if error rate exceeds the configured threshold
6. Posts a deployment summary to Slack and closes the Jira ticket

6.3 Configuration: Tuning the Pipeline

Every stage of the pipeline is configurable through CLAUDE.md and skill configuration files. Key parameters that teams typically tune:

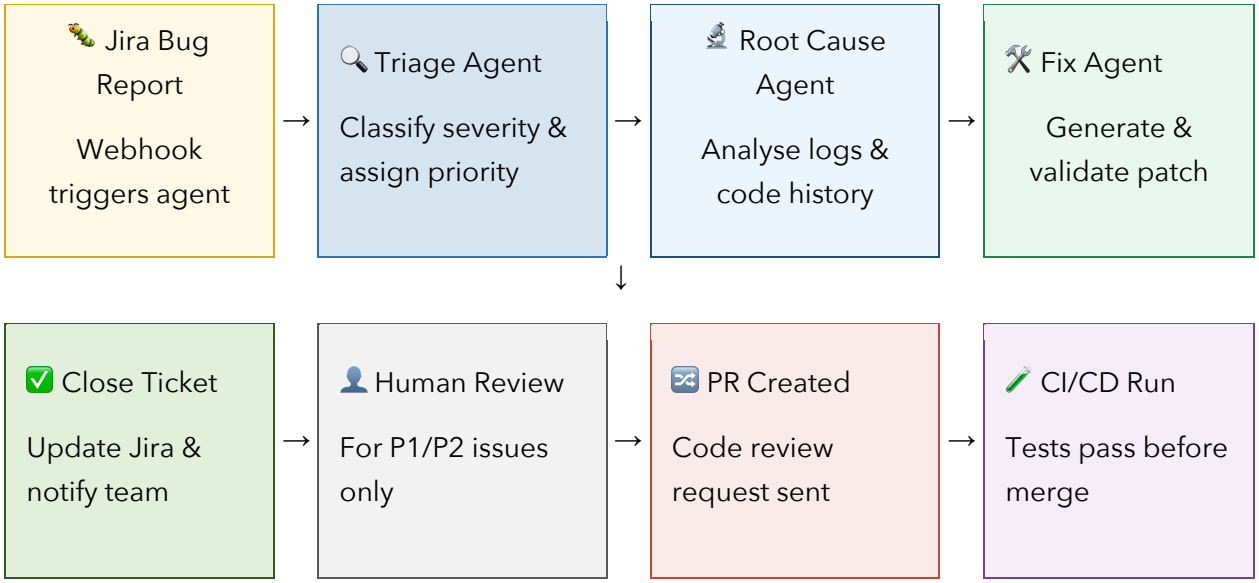
Key Configuration Parameters	**Details**
HUMAN_APPROVAL_THRESHOLD	Story points or complexity score above which human review is required
MIN_TEST_COVERAGE	Minimum coverage % before a PR is allowed (default: 80)
AUTO_DEPLOY_TO_STAGING	true/false: deploy to staging automatically after PR merge
CANARY_DURATION_MINUTES	How long to monitor a canary before promoting to 100%
ROLLBACK_ERROR_RATE	Error rate % that triggers automatic rollback (default: 1.0)
ESCALATION_SLACK_CHANNEL	Where to post escalations requiring human decision
MAX_AGENT_ITERATIONS	Safety limit on agent loop iterations per task

7. Workflow 2: Automated Bug Resolution

7.1 Overview

This workflow demonstrates a fully automated pipeline triggered by an external event – a bug report arriving in Jira. No human initiates anything. The system triages, investigates, resolves, and (for lower-priority issues) closes the ticket without human involvement. For high-priority issues, humans are pulled in at exactly the right moment with all the context they need.

This pattern is ideal for: flaky test failures, recurring known issues, low-complexity regressions, and configuration-related errors that follow predictable patterns.



7.2 Trigger: Jira Webhook

The pipeline is activated by a Jira webhook. When a bug report is created or transitioned to 'To Do', Jira sends a POST request to a lightweight webhook handler. This handler validates the payload and invokes the Triage Agent via Claude Code.

```
# Webhook handler (simplified Python)
from flask import Flask, request
import subprocess

app = Flask(__name__)

@app.route('/jira-webhook', methods=['POST'])
def handle_jira_event():
    payload = request.json
```

```
    if payload['issue']['fields']['issuetype']['name'] == 'Bug':
        issue_key = payload['issue']['key']
        subprocess.Popen([
            'claude',                    # Claude Code CLI
            '--skill', 'skills/bug-triage/SKILL.md',
            '--input', f'jira_issue={issue_key}',
            '--max-turns', '25',
        ])
    return {'status': 'accepted'}, 202
```

7.3 Stage-by-Stage Walkthrough

Stage 1: Triage

The Triage Agent reads the Jira ticket, any attached logs, and the error description. It classifies the bug:

Priority Classification Rules (configurable in bug-triage/SKILL.md)
P1 (Critical) – Production down or data loss. Always escalate to human immediately.
P2 (High) – Major feature broken, no workaround. Human review before deploying fix.
P3 (Medium) – Feature degraded, workaround exists. Auto-fix, human reviews PR.
P4 (Low) – Minor cosmetic or UX issue. Fully automated: fix, test, merge, close.
Classification is based on: affected_users, error_type, revenue_impact, recurrence_rate.

Stage 2: Root Cause Analysis

The Root Cause Agent uses the Debug skill to investigate. It:

1. Fetches the last 500 lines of application logs around the time of the error
2. Queries the distributed tracing system for the failed request's trace
3. Searches Git history for recent changes to affected files
4. Cross-references with similar past bugs in Jira to find patterns
5. Produces a Root Cause Report, citing specific files, line numbers, and commits

Stage 3: Fix Generation

Armed with the root cause analysis, the Fix Agent:

1. Creates a fix branch from main
2. Implements the minimal change needed to resolve the bug

3. Adds a regression test that would have caught this bug
4. Runs the full test suite to ensure no regressions
5. Attaches the root cause report and fix description to the Jira ticket

Stage 4: Human Review (P1/P2 only)

For P1 and P2 issues, the system pauses and notifies the on-call engineer via PagerDuty or Slack with a pre-built context package:

- What broke and when
- Root cause with evidence (log lines, traces, offending commit)
- The proposed fix with a diff
- Test results showing the fix works
- One-click approval link

The human can approve (triggering immediate deployment), request changes (returning to the Fix Agent), or reject and take manual control. For P3/P4 issues, this stage is skipped entirely.

Stage 5: Deployment & Close

After approval (or immediately for P3/P4), the fix is merged, deployed, and validated using the same canary mechanism described in Workflow 1. Once the error rate normalises, the agent:

- Transitions the Jira ticket to Done
- Posts a resolution summary with time-to-resolve metrics
- Updates the runbook if a new class of bug was encountered

7.4 Why This Works: Reliability Through Specialisation

The automated bug workflow is reliable precisely because each agent is specialised. The Triage Agent does not attempt to fix bugs – it focuses on accurate classification. The Root Cause Agent does not generate code – it focuses on evidence gathering. This separation prevents agents from overstepping their competence and makes the system easier to audit, improve, and debug.

When a workflow fails (e.g. the Root Cause Agent cannot find a clear cause), it escalates with everything it found, rather than guessing. Human escalation is a designed, graceful fallback – not an error condition.

8. Human-in-the-Loop Design

8.1 Principles

Effective human-in-the-loop design is not about maximising human touchpoints – that defeats the purpose of automation. It is about placing human judgment exactly where it adds the most value:

- At decision points with high uncertainty or high consequence
- Where domain knowledge or business context cannot be encoded in a skill
- Where errors are hard to reverse (deployments, data migrations, security changes)
- Where organisational accountability requires a named human approver

8.2 Escalation Conditions

Each skill file defines its own escalation conditions. Common patterns:

Condition	Escalation Action
Agent confidence below threshold	Post draft to Slack for human review before proceeding
Task exceeds configured complexity	Request architecture review before design is finalised
Security-sensitive code changed	Mandatory security engineer sign-off
Test coverage drops	Block PR until coverage is restored (auto or human fix)
Production error rate spike	Page on-call engineer immediately
Agent stuck after N retries	Hand off to human with full context summary
Conflicting requirements detected	Request clarification from the product owner

8.3 The Approval Interface

When human approval is needed, agents produce a structured approval request – not a wall of text. A good approval request answers: What is being done, why, what are the risks, and what happens if the human approves vs. rejects. Agents should make approval easy: provide a one-sentence TL;DR, a diff link, and a clear approve/reject/comment action.

9. Configuring & Evolving the System

9.1 Getting Started: The Minimum Viable Configuration

You do not need to build all agents and skills at once. A recommended progression:

1. Start with a single Code Agent and a code-review skill on one repository
2. Add a Test Agent once you trust the Code Agent's output quality
3. Introduce the Triage Agent for one class of bugs (e.g., test failures)
4. Gradually increase automation scope as confidence grows
5. Add the Orchestrator only when you have 3+ specialist agents to coordinate

9.2 Versioning Skills

Skills are text files – treat them like code. Store skills in a dedicated directory in your repository (e.g. .claude/skills/), version-control them, and review changes via PR. When you update a skill, run it against a set of reference tasks to confirm it still performs correctly. This is called skill evaluation and is as important as unit testing.

9.3 Measuring Workflow Quality

Track these metrics to understand how your agentic workflows are performing:

- Task completion rate: What % of tasks are completed without human intervention?
- Escalation rate: Are too many tasks being escalated (skills too cautious) or too few (skills overconfident)?
- Time-to-resolve: How does agentic resolution compare to manual resolution for bugs?
- False positive rate: How often does the agent raise false alarms or incorrect diagnoses?
- Human approval rate: What fraction of agent-generated PRs are approved first time?

9.4 Safety & Guardrails

Claude Code has built-in safety mechanisms that operators can configure:

- Allowed tool list: Explicitly whitelist which tools each agent may call
- Filesystem boundaries: Restrict write access to specific directories
- Network access controls: Prevent agents from calling external services not in the allowlist
- Action confirmation: Require human confirmation for destructive actions (delete, deploy, send)
- Token and iteration limits: Prevent runaway agents from consuming excessive resources

These controls are configured in CLAUDE.md and .claude/settings.json, giving operators fine-grained control without modifying agent logic.

10. Practical Implementation Guide

10.1 Repository Structure

A recommended repository structure for a project using multi-agent workflows:

```
myproject/
├── CLAUDE.md                  # Top-level Claude Code configuration
├── .claude/
│   ├── settings.json          # Safety & tool access controls
│   └── mcp_settings.json      # MCP server connections
├── skills/
│   ├── requirements/SKILL.md
│   ├── architecture/SKILL.md
│   ├── code-generation/SKILL.md
│   ├── test-generation/SKILL.md
│   ├── code-review/SKILL.md
│   ├── bug-triage/SKILL.md
│   └── deployment/SKILL.md
├── agents/
│   ├── orchestrator.md        # Orchestrator agent instructions
│   └── specialist/            # Per-specialist overrides
├── src/                       # Application source code
└── tests/                     # Test suite
```

10.2 Installing Claude Code

Claude Code is available as an npm package and runs on macOS, Linux, and Windows (WSL). Installation:

```
# Install Claude Code globally
npm install -g @anthropic-ai/claude-code

# Authenticate with your Anthropic API key
export ANTHROPIC_API_KEY=your_api_key_here
```

```
# Verify installation
claude --version

# Run Claude Code in your project directory
cd myproject
claude
```

10.3 Running Your First Agent Task

Once installed, you can invoke Claude Code with a skill and task:

```
# Run the bug-triage skill on a specific Jira issue
claude \
  --skill skills/bug-triage/SKILL.md \
  --input 'jira_issue=MYAPP-1234' \
  --max-turns 20 \
  --output-format json

# Run the full development pipeline from a requirement
claude \
  --skill skills/requirements/SKILL.md \
  --input 'requirement=Add PDF export to the reports module' \
  --max-turns 50
```

11. Common Patterns & Anti-Patterns

11.1 Patterns That Work Well

- **Narrow skills**: A skill that does one thing well is more reliable than a skill that tries to do everything. Resist the temptation to create a 'super skill'.
- **Explicit outputs**: Skills that specify their output format (JSON schema, file path, Jira comment) are easier to chain together and easier to test.
- **Graceful escalation**: Every skill should have a well-defined 'I don't know' path that hands off to a human with full context rather than guessing.
- **Idempotent actions**: Design agent actions so they can be safely retried without side effects. Check before creating (e.g. 'does this branch already exist?').
- **Incremental automation**: Start with agents that assist humans (draft code, suggest tests), then progressively automate more as trust is established.

11.2 Anti-Patterns to Avoid

- **God-agent**: One agent that does requirements, design, code, tests, and deployment. Hard to debug, hard to improve, and prone to compounding errors.
- **No escalation path**: An agent that always tries to complete a task, even when it is clearly stuck or out of its depth. This leads to confident but wrong outputs.
- **Unversioned skills**: Editing SKILL.md files without version control means you cannot roll back when a change degrades performance.
- **Hardcoded secrets**: Never put API tokens, passwords, or keys in skill files or CLAUDE.md. Use environment variables or a secrets manager.
- **Skipping the evaluation phase**: Deploying a new skill without testing it against reference tasks is equivalent to deploying code without tests.

12. Future Directions

The field is evolving rapidly. Several developments will shape how agentic development workflows mature:

Persistent Agent Memory

Current agents largely start fresh with each task, relying on context provided in the prompt. Future systems will maintain persistent memory across sessions — remembering that 'we always use the repository pattern in this codebase' or 'this developer prefers minimal PR descriptions'. This will reduce configuration burden and make agents feel more like team members than tools.

Multi-Agent Collaboration

As agent frameworks mature, multiple agents will collaborate more fluidly — reviewing each other's work, negotiating on ambiguous decisions, and specialising dynamically based on the task at hand. Standards like MCP will make it easier to connect agents from different providers.

Agent Evaluation & Benchmarking

Systematic evaluation of agent performance — analogous to unit testing — will become standard practice. Teams will maintain benchmark task suites and measure skill quality the same way they measure code quality today.

Regulatory & Compliance Integration

As agents make more decisions autonomously, audit trails and explainability become regulatory requirements in some industries. Expect agent frameworks to build in compliance logging, approval chains, and report generation as first-class features.

13. Conclusion

AI Agents and Skills represent a fundamental shift in how software is built – not by replacing developers, but by giving them a team of tireless, knowledgeable collaborators. When designed well, these systems handle the mechanical and repetitive aspects of development, freeing humans to focus on the creative, strategic, and interpersonal work that machines cannot do.

The key principles to carry forward:

1. Specialise agents: narrow focus, clear inputs, explicit outputs
2. Skills are code: version them, test them, review changes carefully
3. Humans belong at decision points, not bottlenecks
4. Start small and earn trust incrementally
5. Configuration is the primary control surface – invest in CLAUDE.md and skill files
6. Escalation is a feature, not a failure

Based on my experiende, I believe that Claude Code provides a mature, safe, and extensible runtime for building these workflows today. The architecture described in this paper – orchestrator, specialists, skills, MCP tools – is not a future vision. It is deployable now, and teams around the world are already using it to compress delivery timelines, improve code quality, and reduce the cognitive load on their engineering teams.

The most effective teams will be those that treat their agents and skills with the same engineering rigour they apply to their application code: designed thoughtfully, tested thoroughly, monitored continuously, and improved iteratively.

Appendix A: Terminology

Term	Definition
Agent	An AI system that autonomously plans and executes multi-step tasks using tools, memory, and reasoning.
Skill	A reusable, self-contained instruction set that teaches an agent how to perform a specific task.
Orchestrator	An agent responsible for decomposing top-level goals and coordinating specialist agents.
MCP (Model Context Protocol)	An open standard by Anthropic for connecting agents to external tools and services.
CLAUDE.md	A project-level configuration file that tells Claude Code about conventions, available skills, and workflow rules.
SKILL.md	A Markdown file that defines a single agent skill: objective, inputs, process, outputs, and configuration.
ReAct	Reasoning + Acting: the observe-think-act loop that most LLM-based agents implement.
Human-in-the-loop	Design pattern where human review and approval is integrated at specific workflow stages.
Canary Deployment	A deployment pattern where a change is initially rolled out to a small percentage of users before full release.
ADR	Architecture Decision Record: a document capturing an important architectural decision, its context, and consequences.

Appendix B: Further Reading

- Anthropic Claude Code Documentation: https://docs.anthropic.com/claude/claude-code
- Model Context Protocol Specification: https://modelcontextprotocol.io
- Anthropic Prompt Engineering Guide: https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering
- ReAct: Synergizing Reasoning and Acting in Language Models (Yao et al., 2023)
- Building Effective Agents – Anthropic Blog: https://www.anthropic.com/research/building-effective-agents