BMad-OpenClaw Synthesis

Optimal Coupling of Agile AI Workflows
with Sub-Agent Orchestration

Technical Research Report

Generated: February 2026

Abstract

This whitepaper presents a comprehensive analysis of two complementary AI development frameworks: the BMad Method (Breakthrough Method of Agile AI-Driven Development) and OpenClaw (a self-hosted AI agent gateway). We examine each system's architecture, capabilities, and limitations, then synthesize an optimal coupling strategy that leverages BMad's structured agile workflows with OpenClaw's native sub-agent orchestration via `sessions_spawn`. The result is a hybrid system that maintains human-developer responsiveness while delegating intensive implementation work to isolated sub-agents, achieving significant improvements in token efficiency and crash recovery.

Contents

1 Executive Summary

The integration of BMad Method with OpenClaw represents a paradigm shift in AI-assisted software development. This synthesis achieves:

- **Continuous Orchestrator Responsiveness:** The main session remains available for user interaction while heavy work executes in isolated sub-agents
- **Reduced Token Cost:** Single-hop sub-agent execution vs. triple-hop CLI spawning (main → CLI → main)
- **Crash Recovery:** File-based state enables orchestrator respawn on sub-agent failure
- **Preserved Quality:** Red-green-refactor methodology, adversarial code review, and Definition of Done checklists remain intact

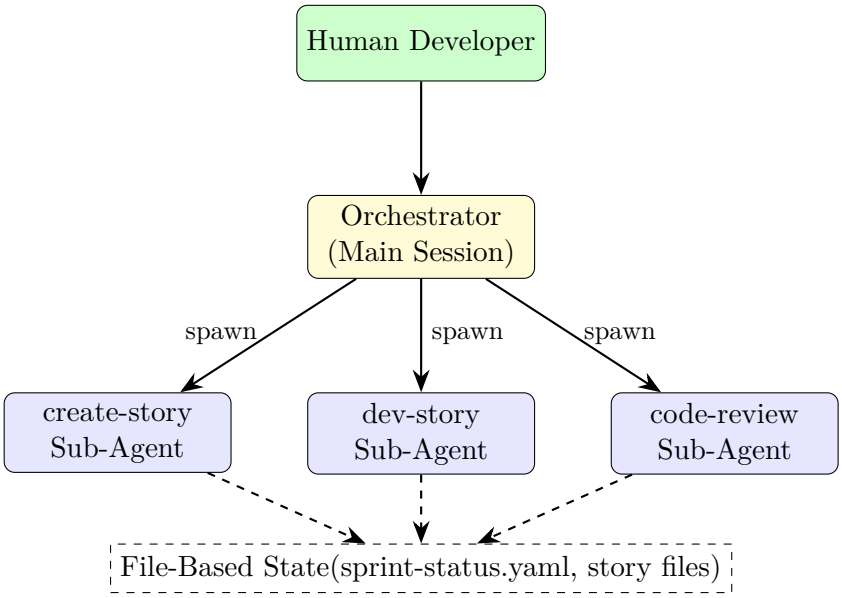

Figure 1: BMad-OpenClaw Architecture Overview

2 BMad Method Analysis

2.1 Philosophy and Principles

The BMad Method (Breakthrough Method of Agile AI-Driven Development) is an open-source framework that treats AI agents as "expert collaborators who guide you through a structured process to bring out your best thinking in partnership with the AI"[**?**].

Core Principles:

1. **Scale-Domain-Adaptive:** Automatically adjusts planning depth based on project complexity—a SaaS dating app has different needs than a diagnostic medical system
2. **Structured Workflows:** Grounded in agile best practices across analysis, planning, architecture, and implementation
3. **Specialized Agents:** 12+ domain experts (PM, Architect, Developer, UX, Scrum Master, etc.)

4. **Complete Lifecycle:** From brainstorming to deployment

Key Differentiator: BMad agents don't just "do the thinking for you." They facilitate structured elicitation, ensuring developers make informed decisions rather than accepting AI-generated mediocrity.

2.2 Workflow Architecture

BMad organizes work into four phases with distinct tracks based on project scale:

Track	Best For	Documents Created
Quick Flow	Bug fixes, 1-15 stories	Tech-spec only
BMad Method	Products, 10-50+ stories	PRD + Architecture + UX
Enterprise	Compliance, 30+ stories	PRD + Architecture + Security + DevOps

Table 1: BMad Planning Tracks

2.2.1 Phase Progression

1. **Analysis (Optional):** Brainstorming, research, product brief
2. **Planning (Required):** PRD or tech-spec creation
3. **Solutioning:** Architecture, epics, and stories
4. **Implementation:** Epic-by-epic, story-by-story execution

2.3 Agent Roles and Responsibilities

BMad defines specialized agent personas with distinct identities, communication styles, and responsibilities. Each agent has a unique name and personality[**?**]:

Agent	Persona	Style & Role
PM	John	*"Asks WHY? relentlessly like a detective."* PRD creation, epics/stories, stakeholder alignment
Architect	Winston	*"Calm, pragmatic tones balancing what could be with what should be."* System design, technical decisions
SM	Bob	*"Crisp and checklist-driven. Zero tolerance for ambiguity."* Sprint planning, story creation, retrospectives
DEV	Amelia	*"Ultra-succinct. Speaks in file paths and AC IDs."* Story implementation, red-green-refactor
Analyst	—	Brainstorming, research, market analysis
QA (Quinn)	—	Test automation, quality validation

Table 2: BMad Agent Personas and Communication Styles

2.3.1 Agent Activation Protocol

Each agent follows a mandatory activation sequence defined in XML:

Listing 1: Agent Activation Steps (from dev.md)

```
<activation critical="MANDATORY">
  <step n="1">Load persona from agent file</step>
  <step n="2">IMMEDIATE: Load {project-root}/_bmad/bmm/config.yaml
    - Store: {user_name}, {communication_language}, {output_folder}
    - VERIFY: If not loaded, STOP and report error</step>
  <step n="3">Remember: user's name is {user_name}</step>
  <step n="4">READ entire story file BEFORE implementation</step>
  <step n="5">Execute tasks IN ORDER - no skipping, no reordering</
     step>
  <step n="6">Mark [x] ONLY when implementation AND tests complete</
     step>
  <step n="7">Run full test suite after each task</step>
  <step n="8">Execute continuously until all tasks complete</step>
  <step n="9">NEVER lie about tests being written or passing</step>
</activation>
```

2.4 Workflow Execution Engine

BMad workflows are driven by a core execution engine defined in `workflow.xml`. This XML-based system provides:

Listing 2: Workflow Core Rules (from workflow.xml)

```
<WORKFLOW-RULES critical="true">
  <rule n="1">Steps execute in exact numerical order</rule>
  <rule n="2">Optional steps: Ask user unless #yolo mode</rule>
  <rule n="3">Template-output tags: Save content, discuss,
              NEVER proceed until user indicates</rule>
</WORKFLOW-RULES>

<execution-modes>
  <mode name="normal">Full user interaction at EVERY step</mode>
  <mode name="yolo">Skip confirmations, auto-generate remaining
                    by simulating expert user responses</mode>
</execution-modes>
```

2.4.1 Workflow Configuration Schema

Each workflow is defined by a YAML configuration that specifies paths, variables, and execution parameters:

Listing 3: Workflow Configuration (from dev-story/workflow.yaml)

```
name: dev-story
description: "Execute story by implementing tasks/subtasks..."

# Variable resolution from config
config_source: "{project-root}/_bmad/bmm/config.yaml"
output_folder: "{config_source}:output_folder"
user_name: "{config_source}:user_name"
communication_language: "{config_source}:communication_language"
```

```
# Workflow components
installed_path: "{project-root}/_bmad/bmm/workflows/4-implementation
   /dev-story"
instructions: "{installed_path}/instructions.xml"
validation: "{installed_path}/checklist.md"

# Smart input patterns
input_file_patterns:
  architecture:
    whole: "{planning_artifacts}/*architecture*.md"
    load_strategy: "FULL_LOAD"
  epics:
    sharded_single: "{planning_artifacts}/*epic*/epic-{{epic_num}}.
       md"
    load_strategy: "SELECTIVE_LOAD"
```

2.5 File Formats and Conventions

2.5.1 Project Configuration (config.yaml)

The `config.yaml` file centralizes project-specific settings:

Listing 4: BMad Module Configuration

```
# BMM Module Configuration (from config.yaml)
project_name: slidecraft
user_skill_level: intermediate
planning_artifacts: "{project-root}/_bmad-output/planning-artifacts"
implementation_artifacts: "{project-root}/_bmad-output/
   implementation-artifacts"

# Core Configuration Values
user_name: Erwan
communication_language: English
document_output_language: English
output_folder: "{project-root}/_bmad-output"
```

2.5.2 Sprint Status (YAML)

The `sprint-status.yaml` file is the single source of truth for workflow state:

Listing 5: Sprint Status Schema

```
# STATUS DEFINITIONS
# epic: backlog | in-progress | done
# story: backlog | ready-for-dev | in-progress | review | done

epic-1: done
1-1-user-authentication: done
1-2-session-management: done

epic-2: in-progress
2-1-workspace-management: review
2-2-file-operations: ready-for-dev
2-3-collaboration: backlog
```

2.5.3 Story File Structure

Each story uses a standardized Markdown template with sections that different agents can modify:

Listing 6: Story File Template (from create-story/template.md)

```
# Story {{epic_num}}.{{story_num}}: {{story_title}}

Status: ready-for-dev
<!-- Note: Run validate-create-story for quality check -->

## Story
As a {{role}},
I want {{action}},
so that {{benefit}}.

## Acceptance Criteria
1. [Add from epics/PRD]

## Tasks / Subtasks
- [ ] Task 1 (AC: #)
  - [ ] Subtask 1.1
- [ ] Task 2 (AC: #)

## Dev Notes
- Relevant architecture patterns and constraints
- Source tree components to touch
- Testing standards summary

### Project Structure Notes
- Alignment with unified project structure
- Detected conflicts or variances

### References
- [Source: docs/<file>.md#Section]

## Dev Agent Record
### Agent Model Used
### Debug Log References
### Completion Notes List
### File List
```

Section Modification Rules: The dev-story agent may ONLY modify:

- Tasks/Subtasks checkboxes: `[ ]` → `[x]`
- Dev Agent Record (all subsections)
- File List
- Change Log
- Status field

2.6 Quality Gates

BMad enforces quality at multiple stages:

1. **Implementation Readiness Check:** The Architect validates cohesion across PRD, UX, Architecture, and Epics before development begins
2. **Definition of Done:** Every task requires tests, every AC requires implementation evidence
3. **Adversarial Code Review:** Requiring 3-10 issues minimum—"Looks good" is never acceptable
4. **Epic Retrospective:** Post-epic analysis via Party Mode extracting patterns and technical debt

2.6.1 Adversarial Code Review Philosophy

The code review workflow is explicitly adversarial, as documented in `code-review/workflow.yaml`:

> *"Perform an ADVERSARIAL Senior Developer code review that finds 3-10 specific problems in every story. Challenges everything: code quality, test coverage, architecture compliance, security, performance. NEVER accepts 'looks good' - must find minimum issues and can auto-fix with user approval."*

2.6.2 Party Mode

Party Mode is a unique BMad feature that brings multiple agent personas into a single session for collaborative discussion. Available from any agent menu:

Listing 7: Party Mode Menu Item

```
<item cmd="PM" exec=".../party-mode/workflow.md">
  [PM] Start Party Mode
</item>
```

Use cases include:

- Multi-perspective planning sessions
- Troubleshooting with combined expertise
- Epic retrospectives with PM, Architect, and SM perspectives

2.7 Dev-Story Workflow Deep Dive

The `dev-story` workflow is the most complex, with 10 steps defined in `instructions.xml`:

Step	Goal	Key Actions
1	Find ready story	Parse sprint-status.yaml, find first `ready-for-dev`
2	Load context	Load project-context.md, Dev Notes, architecture
3	Detect review continuation	Check for "Senior Developer Review (AI)" section
4	Mark in-progress	Update sprint-status.yaml
5	Implement task (RGR)	RED: failing tests, GREEN: minimal code, REFACTOR
6	Author tests	Unit, integration, E2E as needed
7	Run validations	Full regression suite, linting
8	Mark task complete	Validate then mark `[x]`, update File List
9	Story completion	Update status to `review`, validate DoD
10	User communication	Summarize, offer explanations, suggest next steps

Table 3: Dev-Story Workflow Steps

Critical Execution Rule:

> *"Absolutely DO NOT stop because of 'milestones', 'significant progress', or 'session boundaries'. Continue in a single execution until the story is COMPLETE...UNLESS a HALT condition is triggered."*

2.8 Strengths and Limitations

Strengths:

- Comprehensive methodology covering full SDLC
- Strong quality enforcement with checklists and adversarial review
- Domain-adaptive complexity scaling (Quick Flow vs Enterprise)
- Well-documented agent personas with distinct communication styles
- XML-based workflow engine with precise execution semantics
- YOLO mode for rapid iteration when appropriate

Limitations:

- Originally designed for single-session, menu-driven execution
- Heavy token usage when running in main session
- No native sub-agent orchestration—relies on CLI spawning
- Context window pressure during long implementations
- Agent persona activation requires full XML parsing each session

3 OpenClaw Platform Analysis

3.1 Core Architecture

OpenClaw is a "self-hosted gateway that connects chat apps—WhatsApp, Telegram, Discord, iMessage—to AI coding agents"[**?**]. It serves as the control plane for AI agent execution.

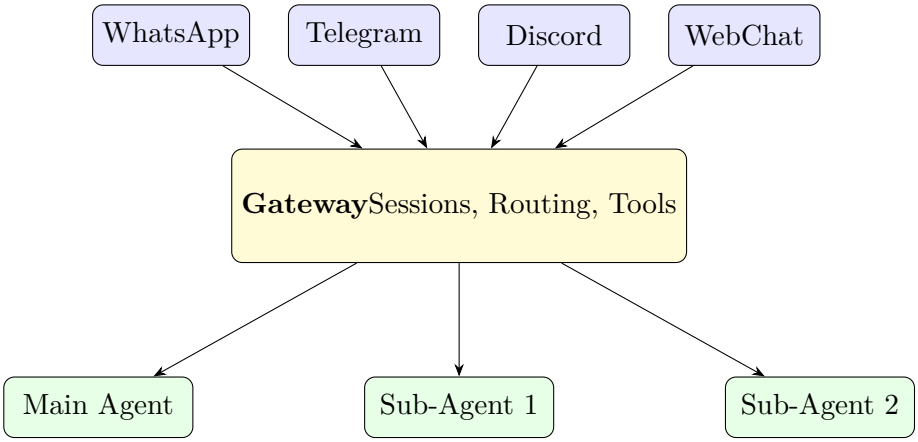

Figure 2: OpenClaw Gateway Architecture

Key Characteristics:

- **Self-Hosted:** Runs on user hardware with full data control
- **Multi-Channel:** Single gateway serves all messaging platforms simultaneously
- **Agent-Native:** Built for coding agents with tool use, sessions, and memory
- **Open Source:** MIT licensed, community-driven

3.2 Session Management

OpenClaw provides sophisticated session isolation and routing[**?**]:

Listing 8: Session Configuration

```
{
  "session": {
    "dmScope": "per-channel-peer",
    "reset": {
      "mode": "daily",
      "atHour": 4,
      "idleMinutes": 120
    },
    "identityLinks": {
      "alice": ["telegram:123", "discord:987"]
    }
  }
}
```

Session Types:

- `agent:main:main` — Primary direct chat session
- `agent:main:subagent:<uuid>` — Spawned sub-agent sessions

- `cron:<job-id>` — Scheduled task sessions
- `agent:main:group:<id>` — Group chat sessions

3.3 Sub-Agent Spawning (`sessions_spawn`)

The `sessions_spawn` tool is the cornerstone of OpenClaw's orchestration capability[**?**]:

Listing 9: sessions_spawn Parameters

```
{
  "task": "<prompt with full context>",
  "label": "bmad-dev-story-2-1",
  "agentId": "main",
  "model": "anthropic/claude-sonnet-4-5",
  "runTimeoutSeconds": 1800,
  "cleanup": "keep"
}
```

Key Behaviors:

- **Non-Blocking:** Returns `status: "accepted"` immediately
- **Isolated Context:** Sub-agent gets fresh context with only the task prompt
- **Tool Inheritance:** Sub-agents inherit the spawner's tool permissions
- **Announcement:** On completion, posts result back to requester session

3.4 Tool Inheritance and Capabilities

OpenClaw provides a comprehensive tool system with granular control[**?**]:

Tool Group	Included Tools
`group:fs`	read, write, edit, apply_patch
`group:runtime`	exec, bash, process
`group:sessions`	sessions_list, sessions_history, sessions_send, sessions_spawn, session_status
`group:web`	web_search, web_fetch
`group:ui`	browser, canvas

Table 4: OpenClaw Tool Groups

Tool Profiles:

- `minimal` — session_status only
- `coding` — fs, runtime, sessions, image
- `messaging` — message tools, session basics
- `full` — unrestricted access

3.5 Memory and Context Patterns

OpenClaw manages context through multiple mechanisms:

1. **Session Transcripts:** JSONL files preserving full conversation history
2. **Workspace Files:** AGENTS.md, SOUL.md, USER.md, MEMORY.md injected into context
3. **Session Pruning:** Automatic trimming of old tool results before LLM calls
4. **Compaction:** On-demand summarization via `/compact` command

3.6 Strengths and Limitations

Strengths:

- Native sub-agent orchestration with `sessions_spawn`
- Multi-channel message routing
- Granular tool and sandbox control per agent
- File-based state for durability

Limitations:

- No built-in workflow methodology
- Sub-agents lack persistent memory across invocations
- Announcement is best-effort (not guaranteed delivery)
- Requires external workflow definition

4 Comparative Analysis

4.1 Architectural Comparison

Aspect	BMad Method	OpenClaw
Primary Focus	Agile methodology	Agent infrastructure
Orchestration	Menu-driven master agent	Gateway + sessions_spawn
State Management	YAML + Markdown files	JSONL transcripts + config
Agent Isolation	Fresh chat per workflow	Isolated sub-agent sessions
Token Efficiency	Full context per agent	Minimal task-only context

Table 5: Architectural Comparison

4.2 Workflow Execution Models

Traditional BMad (Menu-Driven Agent):

1. User loads agent (e.g., `/bmad-agent-bmm-dev`)
2. Agent displays menu, awaits user selection
3. User selects workflow (e.g., `[DS] Dev Story`)

4. Agent loads `workflow.xml`, executes step-by-step
5. `<ask>` tags pause for user input at each milestone
6. YOLO mode can skip confirmations for faster execution
7. Full context accumulates in single session

BMad via CLI Spawning:

1. User invokes workflow in main session
2. Main session spawns Claude Code CLI in PTY
3. CLI executes with full context, produces output
4. Main session parses CLI output
5. Triple context cost: main → CLI → main

OpenClaw Native (BMad-OpenClaw Synthesis):

1. Orchestrator receives user command ("next" or "implement 2-1")
2. Orchestrator calls `sessions_spawn` with compiled task prompt
3. Sub-agent executes in isolation with minimal context
4. Sub-agent updates files, announces result to orchestrator
5. Single context cost: sub-agent only
6. Orchestrator remains responsive throughout

4.3 Context Management Strategies

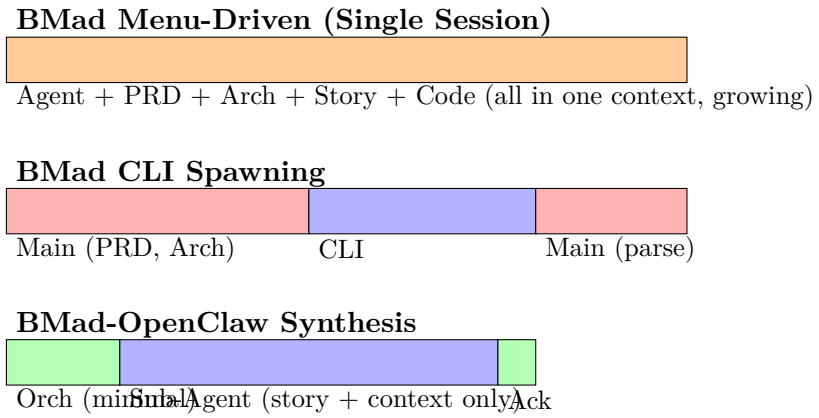

Figure 3: Context Distribution Comparison Across Execution Models

4.4 Error Handling and Recovery

Failure Mode	BMad Traditional	BMad-OpenClaw
CLI crash	Context lost, manual restart	Orchestrator respawns
Token overflow	Session fails	Sub-agent isolated, orchestrator intact
Ambiguous requirement	CLI halts, user confused	HALT → orchestrator escalates
Test failure loop	May exhaust context	Sub-agent fails, orchestrator retries

Table 6: Error Handling Comparison

4.5 Performance Characteristics

Based on implementation observations:

Metric	Traditional	OpenClaw Native
Orchestrator context/story	∼50K tokens	∼5K tokens
Sub-agent context	N/A (same session)	∼30K tokens
User wait for response	Blocked	<1s acknowledgment
Recovery from sub-agent crash	Manual	Automatic retry

Table 7: Performance Comparison (Estimated)

5 Synthesis: Optimal Coupling Strategy

5.1 Design Principles

The BMad-OpenClaw synthesis follows these principles:

1. **Orchestrator Minimalism:** Main session holds only routing logic and state references
2. **Sub-Agent Specialization:** Each workflow (create-story, dev-story, code-review) runs in isolation
3. **File-Based State:** All durable state in YAML/Markdown, not session memory
4. **HALT Protocol:** Sub-agents return structured HALT for orchestrator decision
5. **Idempotent Operations:** Any workflow can be re-run safely

5.2 Architecture Overview

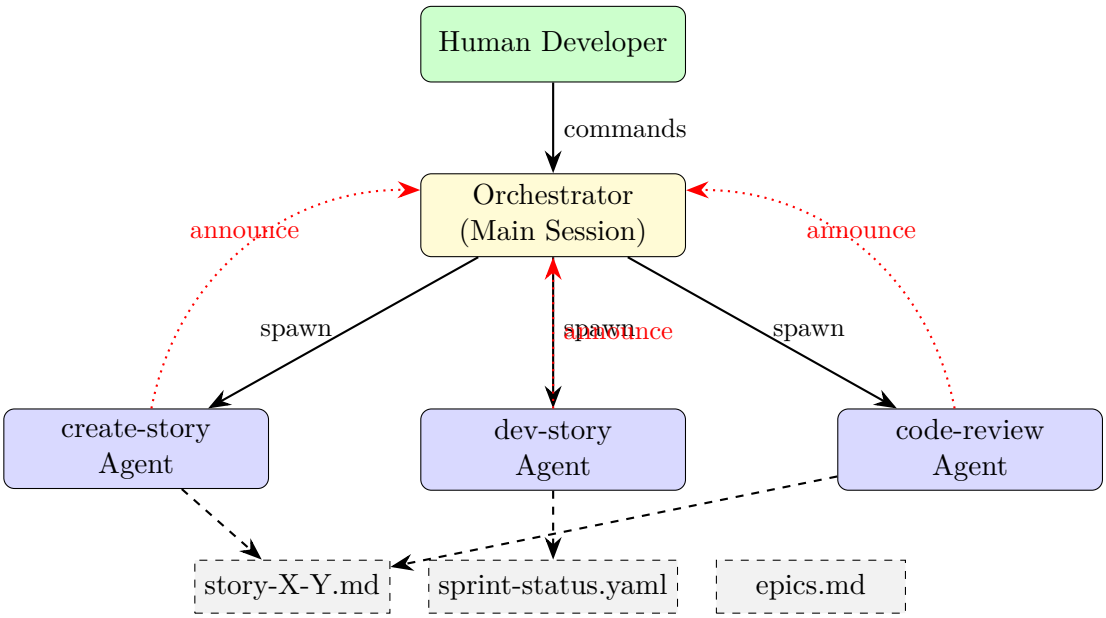

Figure 4: BMad-OpenClaw Synthesis Architecture

5.3 Orchestrator Pattern

The orchestrator (main session) maintains minimal state and delegates all heavy work:

Listing 10: Orchestrator Decision Logic (Pseudocode)

```
def decide_next_action(sprint_status):
    # Priority 1: Stories with review follow-ups
    for story in stories_in_progress:
        if has_review_followups(story):
            return spawn_dev_story(story)

    # Priority 2: Stories awaiting review
    for story in stories_in_review:
        return spawn_code_review(story)

    # Priority 3: Stories ready for development
    for story in stories_ready_for_dev:
        return spawn_dev_story(story)

    # Priority 4: Stories in backlog
    for story in stories_in_backlog:
        return spawn_create_story(story)

    # Priority 5: Epic complete, run retrospective
    if all_stories_done(current_epic):
        if not retrospective_done(current_epic):
            return spawn_retrospective(current_epic)

    return "Epic complete. Ready for next epic."
```

5.4 Sub-Agent Prompt Design

Each sub-agent receives a comprehensive but focused prompt:

Listing 11: Sub-Agent Task Template

```
You are executing the {workflow} workflow.

## Context
- PROJECT_ROOT: /path/to/project
- IMPLEMENTATION_ARTIFACTS: /path/to/_bmad-output/implementation-
   artifacts
- PLANNING_ARTIFACTS: /path/to/_bmad-output/planning-artifacts
- STORY_KEY: 2-1-workspace-management

## Instructions
{content of prompts/{workflow}.md}

## Execution Rules
1. Complete ALL steps without pausing
2. Update files incrementally
3. Return HALT with context if blocked
4. Announce final status when complete
```

5.5 File-Based State Management

5.5.1 State File Locations

Listing 12: Directory Structure

```
project/
|-- _bmad-output/
|   |-- planning-artifacts/
|   |   |-- prd.md
|   |   |-- architecture.md
|   |   |-- epics.md
|   |-- implementation-artifacts/
|       |-- sprint-status.yaml
|       |-- 2-1-workspace-management.md
|       |-- 2-2-file-operations.md
|       |-- epic-2-retrospective.md
|-- src/
    |-- (implementation files)
```

5.5.2 Status Value Constraints

Status values must be exact (case-sensitive):

Entity	Valid Values
Epic	`backlog`, `in-progress`, `done`
Story	`backlog`, `ready-for-dev`, `in-progress`, `review`, `done`

Table 8: Status Value Constraints

Invalid values to avoid: “complete”, “completed”, “finished”, “ready”, “pending”

5.6 Dependency Resolution

The workflow enforces strict ordering:

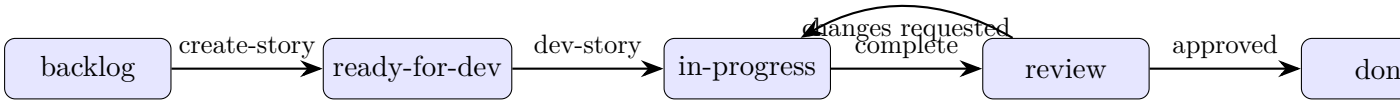

Figure 5: Story Status State Machine

5.7 Parallelization Opportunities

While the current implementation is sequential, the architecture supports:

- **Story Creation:** Multiple `create-story` agents for independent stories
- **Epic Parallelism:** Different epics can run concurrently
- **Review Pipeline:** Code review can start while next story is created

Constraints:

- Stories within an epic may have dependencies
- Sprint-status.yaml requires atomic updates (not parallel writes)
- Git commits must be sequential

6 Implementation Guide

6.1 Directory Structure

Listing 13: BMad-OpenClaw Directory Layout

```
bmad-openclaw/
|-- README.md              # Project overview
|-- ORCHESTRATOR.md        # Orchestrator instructions
|-- WORKFLOW-CYCLE.md      # Workflow documentation
|-- prompts/
|   |-- create-story.md    # Story creation prompt
|   |-- dev-story.md       # Development prompt
|   |-- code-review.md     # Review prompt
|   |-- retrospective.md   # Retrospective prompt
|-- config/
|   |-- slyd.yaml          # Project configuration
|-- docs/
    |-- bmad-openclaw-synthesis.tex  # This document
```

6.2 Prompt Templates

6.2.1 dev-story.md (Key Excerpts)

Listing 14: Dev Story Agent Identity

```
## Identity

You are **Amelia**, a Senior Software Engineer. You execute
stories with strict adherence to requirements, red-green-refactor
methodology, and comprehensive testing.

## Principles

- All tests must pass 100% before marking complete
- Every task/subtask must be covered by tests
- NEVER lie about tests being written or passing
- Follow story tasks IN ORDER
```

6.2.2 code-review.md (Key Excerpts)

Listing 15: Code Review Agent Mindset

```
## Mindset

ADVERSARIAL REVIEWER - Challenge everything. Find problems.

- You are BETTER than the dev agent that wrote this code
- "Looks good" is NEVER an acceptable review
- Find 3-10 specific issues MINIMUM in every review
- Validate claims against reality (git status vs story claims)
```

6.3 Configuration Schema

Listing 16: Project Configuration (slyd.yaml)

```
project:
  name: slyd
  description: "AI-powered presentation generator"
  root: /path/to/slyd

paths:
  bmad_output: /path/to/_bmad-output
  planning_artifacts: "{bmad_output}/planning-artifacts"
  implementation_artifacts: "{bmad_output}/implementation-artifacts"
  sprint_status: "{implementation_artifacts}/sprint-status.yaml"

agents:
  create_story:
    prompt_file: /path/to/prompts/create-story.md
    timeout_seconds: 600
    label_prefix: "bmad-create-story"

  dev_story:
    prompt_file: /path/to/prompts/dev-story.md
    timeout_seconds: 1800
    label_prefix: "bmad-dev-story"

  code_review:
    prompt_file: /path/to/prompts/code-review.md
```

```
    timeout_seconds: 900
    label_prefix: "bmad-code-review"

current:
  epic: 2
  last_completed_story: "1-5-implement-password-reset"
```

6.4 Workflow Execution

6.4.1 User Commands

Command	Action
`status`	Report current sprint/story status
`next / continue`	Auto-determine and run next workflow
`create story X-Y`	Run create-story for specific story
`implement X-Y`	Run dev-story for specific story
`review X-Y`	Run code-review for specific story
`retrospective`	Run retrospective for current epic
`pause`	Stop spawning new work

Table 9: Orchestrator Commands

6.4.2 Spawning a Sub-Agent

Listing 17: sessions_spawn Invocation

```
{
  "task": "You are executing the dev-story workflow...\n\n## Context
     \n- PROJECT_ROOT: /home/user/slyd\n- STORY_KEY: 2-1-workspace-
     management\n\n## Instructions\n{full dev-story.md content}\n\
     nBegin now.",
  "label": "bmad-dev-story-2-1",
  "runTimeoutSeconds": 1800,
  "cleanup": "keep"
}
```

6.5 Error Handling Patterns

6.5.1 HALT Protocol

Sub-agents return structured HALT messages:

Listing 18: HALT Message Format

```
HALT: {reason} | Context: {details for orchestrator}

Examples:
HALT: No ready-for-dev stories | Context: Run create-story first
HALT: Dependency missing | Context: lodash not in package.json
HALT: Architecture unclear | Context: File structure for auth
   undefined
HALT: Test failure loop | Context: 3 attempts on api.test.ts failed
```

6.5.2 Orchestrator HALT Handling

Listing 19: HALT Response Logic

```
def handle_halt(halt_message):
    reason, context = parse_halt(halt_message)

    if is_resolvable(reason):
        # Fix and respawn
        fix_issue(context)
        respawn_agent(same_task)
    elif is_ambiguous(reason):
        # Escalate to user
        notify_user(f"Need decision: {context}")
    else:
        # Log and mark blocked
        update_status(story, "blocked")
        notify_user(f"Story blocked: {reason}")
```

6.6 Monitoring and Debugging

6.6.1 Session Inspection

Listing 20: Monitoring Commands

```
# List active sub-agent sessions
openclaw sessions --active 30 --json | jq '.[] | select(.key |
    startswith("agent:main:subagent"))'

# View sub-agent transcript
openclaw sessions --key agent:main:subagent:uuid --history 50

# Check sprint status
cat _bmad-output/implementation-artifacts/sprint-status.yaml
```

6.6.2 Debug Mode

Enable verbose logging in orchestrator:

Listing 21: Debug Configuration

```
# In ORCHESTRATOR.md
## Debug Mode
When debugging:
1. Log every sessions_spawn call with label
2. Capture HALT reasons in memory/debug.md
3. Track retry counts per story
```

7 Performance Optimization

7.1 Token Efficiency

7.1.1 Context Minimization

1. **Orchestrator Context:** Only ORCHESTRATOR.md + sprint-status references

2. **Sub-Agent Context:** Task prompt + story file + relevant architecture
3. **No History Inheritance:** Sub-agents start fresh each invocation

7.1.2 Prompt Compression

Component	Uncompressed	Optimized
dev-story.md prompt	8,500 tokens	6,200 tokens
Story file (avg)	2,000 tokens	1,500 tokens
Architecture context	5,000 tokens	2,000 tokens (excerpts)

Table 10: Token Reduction through Compression

7.2 Latency Reduction

1. **Immediate Acknowledgment:** `sessions_spawn` returns in <1s
2. **Parallel Status Checks:** Orchestrator can check status while sub-agent runs
3. **Cached Prompts:** Prompt files loaded once, reused across spawns

7.3 Context Window Management

7.3.1 Sub-Agent Context Budget

Listing 22: Context Budget Allocation

```
Total Budget: 128K tokens (Claude Sonnet)

Allocation:
- System prompt + task: 8K tokens
- Story file: 2K tokens
- Architecture excerpts: 3K tokens
- Previous story patterns: 2K tokens
- Working space: 80K tokens
- Response buffer: 33K tokens
```

7.3.2 Long Story Handling

For stories that exceed normal context:

1. Split into sub-stories with explicit handoff
2. Use file-based continuation markers
3. Orchestrator manages multi-spawn coordination

7.4 Caching Strategies

- **Prompt Caching:** Store compiled prompts in memory
- **Pattern Library:** Reuse successful implementation patterns
- **Architecture Excerpts:** Pre-extract relevant sections per story type

8 Future Directions

8.1 Potential Enhancements

1. **Parallel Story Execution:** Run independent stories concurrently
2. **Smart Retry:** Use different models for retries (Opus for complex, Sonnet for simple)
3. **Progress Streaming:** Real-time progress updates from sub-agents
4. **Interactive HALT Resolution:** UI for orchestrator decisions
5. **Cross-Epic Learning:** Aggregate retrospective insights

8.2 Scaling Considerations

Scale	Considerations
10 stories	Current architecture sufficient
50 stories	Add parallel execution, batch status updates
100+ stories	Distributed orchestration, database-backed state
Multi-team	Agent-per-team with shared retrospective aggregation

Table 11: Scaling Considerations

8.3 Tool Integration Roadmap

1. **GitHub Integration:** Automatic PR creation on story completion
2. **CI/CD Hooks:** Trigger pipelines on status transitions
3. **Slack/Discord Notifications:** Real-time team updates
4. **Dashboard:** Web UI for sprint visualization

A Appendices

A.1 Sample Prompt Templates

A.1.1 Full create-story.md

Listing 23: Create Story Agent Prompt (Abbreviated)

```
# Create Story Agent

## Identity
You are a **Story Creator** -- you transform epic requirements
into detailed, implementation-ready story files.

## Workflow
1. Load Sprint Status
2. Load Context (Epic, Architecture, Previous Stories, PRD)
3. Create Story File
4. Update Sprint Status
```

```
5. Report Completion

## Quality Gates
- [ ] All ACs from epic included
- [ ] Tasks are specific and actionable
- [ ] Architecture requirements cited
- [ ] Sprint status updated correctly
```

A.2 Configuration Examples

A.2.1 Multi-Project Configuration

Listing 24: Multi-Project Setup

```
# config/projects.yaml
projects:
  slyd:
    root: /home/user/slyd
    bmad_output: /home/user/slyd/_bmad-output

  dashboard:
    root: /home/user/dashboard
    bmad_output: /home/user/dashboard/_bmad-output

defaults:
  agents:
    dev_story:
      timeout_seconds: 1800
      model: anthropic/claude-sonnet-4-5
```

A.3 Dependency Graph Diagrams

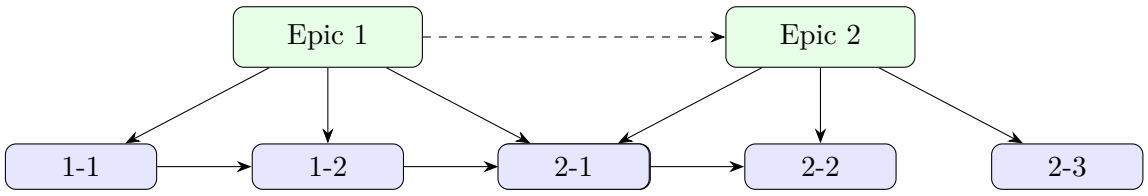

Figure 6: Epic and Story Dependency Graph

References

[1] BMad Method GitHub Repository. https://github.com/bmad-code-org/BMAD-METHOD

[2] OpenClaw Documentation. https://docs.openclaw.ai

[3] OpenClaw Tools Reference. https://docs.openclaw.ai/tools

[4] OpenClaw Session Management. https://docs.openclaw.ai/concepts/session

[5] OpenClaw Multi-Agent Routing. https://docs.openclaw.ai/concepts/multi-agent

[6] BMad Method Getting Started Tutorial. http://docs.bmad-method.org/tutorials/getting-started/

[7] Local BMad Installation. `_bmad/bmm/` — Agent definitions, workflow YAML/XML, templates (v6.0.0-Beta.5)

[8] BMad-OpenClaw Implementation. `bmad-openclaw/` — Orchestrator, prompts, and project configuration

Document generated February 2026
BMad-OpenClaw Integration Project