

Agent Scrum

A sprint methodology for continuous software development by AI agent teams, with observations from 324 sprints

DAVE REMY · WORKING PAPER · VERSION 0.1 · 28 OCTOBER 2026 · CC BY 4.0

ABSTRACT

Over 38 days, a fleet of AI coding agents ran 324 two-hour sprints around the clock on a real product backlog. In the 90 sprints run by a dedicated scrum-team agent, 79% of committed stories reached done inside the sprint and ceremony took under five minutes per sprint. The slowest stage was not writing code: the median time from approval to merge equalled the median time to build. Compounding was the weakest part: 119 of 159 retrospective actions are still open. We call the method that produced these numbers **Agent Scrum**. It keeps Scrum's core devices (a time-boxed sprint, a committed sprint backlog, story points, velocity, a Definition of Done and a retrospective) and changes their scale and constraints. Sprints last two hours instead of two weeks, capacity comes from model rate limits instead of headcount, and cost is measured as *token velocity*, the tokens consumed per merged story point. Conversation and planning stay with one human-facing product-owner agent, while execution moves to scrum-team agents created on demand. The human stakeholder spends under two hours a day, and no sprint waits for them. Horizontal scaling to several teams is designed but untested because of token budget.

KEYWORDS Scrum, agile methods, AI coding agents, multi-agent systems, compound engineering, software factory

1 Introduction

AI coding agents can now take an issue, write the change, test it and open a pull request with little human help. Once a developer has more than one such agent, the question changes from how to prompt an agent to how to organize a group of them. Work has to be chosen, sized, assigned, reviewed, merged and learned from, and the human who owns the result has limited attention.

Software teams have faced this coordination problem before, with people. Scrum is the most widely used answer. This paper describes how a single developer's agent fleet arrived at a version of Scrum by trial and error, which parts carried over unchanged, which needed new meanings, and which did not apply. We call the result **Agent Scrum** and propose it as a methodology others can test.

The author spent years on, and running, human Scrum teams, and came away finding much of it overbearing: a great deal of ceremony, much of which seemed unnecessary for agents. Scrum was therefore not the plan. The first sprints had two motives: to put a boundary around work that otherwise ran without end, and, as much as anything, context management. An agent's context window fills as it works, and quality falls as it fills; a sprint gives the agent a clear, regular point at which to write down its state, clear its context and start fresh. Each later change (points, velocity, a fixed deadline, retrospectives, a separate team) was adopted reluctantly, one at a time, to answer a specific failure. That history matters to the argument: the Scrum devices reappeared because the same problems reappeared, not because we copied a process.

The results were better than that reluctance predicted. Measured against goals most practitioners of applied AI engineering share, the method delivers several of them at once. A software factory that runs around the clock falls out of the sprint loop. Compound engineering falls out of the retrospective. Scrum also brings a large body of existing knowledge (estimation, the time box, the definition of done, the retrospective) that agents and their human can draw on instead of inventing, and its built-in risk management and early feedback carry over intact: small committed batches, a hard boundary, and finished work every two hours. So far, Scrum and agents appear to fit well, and some of the problems that weigh on human Scrum teams have not arisen. It is still early.

The contributions are (1) a description of the method and the mapping from Scrum's roles, artifacts and ceremonies to agent equivalents; (2) a mechanism for enforcing a hard sprint deadline on agents that cannot be interrupted mid-task; (3) measurements from a reference implementation; and (4) an account of open problems.

2 Background: Scrum in brief

This section is for readers who have not worked in Scrum. Readers who have may skip to Section 3. Section 6 lists the terms this paper uses, and Section 11 places Agent Scrum among related work.

Scrum organizes software work into short, fixed periods called **sprints**, traditionally one to four weeks. Before a sprint starts, the team holds **sprint planning**: it takes the highest-priority items from the **product backlog** (the ordered list of everything the product might need) and commits to a subset, the **sprint backlog**. The items are usually written as **user stories**, and a team may split a story into smaller **tasks**.

Each story is estimated in **story points**, a relative measure of effort and risk. A 3-point story is roughly three times the work of a 1-point story. Points have no unit and mean nothing outside the team that assigns them. Their value comes from consistency: over several sprints, the number of points a team completes per sprint, its

velocity, becomes a usable forecast of what it can commit to next time. The ratio of what a team committed to what it finished is often called its **say/do ratio**.

The sprint is **time-boxed**: its end date does not move. Work that is not finished does not extend the sprint; it returns to the backlog. The fixed end forces regular decisions about scope and makes velocity comparable from sprint to sprint. A story counts as finished only when it meets the team's **Definition of Done**, a shared quality bar such as "reviewed, tested and merged".

Scrum has three roles. The **product owner** orders the backlog and decides what is worth building. The **developers** do the work. The **scrum master** runs the process and removes obstacles. It also has a small set of ceremonies: planning, a short daily stand-up for coordination, a **sprint review** that demonstrates the finished work to stakeholders, and a **retrospective** in which the team examines how it worked and chooses improvements. Teams often display the sprint on a **board** with columns such as To do, In progress and Done, a practice borrowed from Kanban.

Two newer terms are relevant. **Compound engineering** describes a practice in which every unit of work leaves the system better at doing the next unit, for example by turning each mistake into a rule, test or tool that prevents it from recurring. A **software factory** describes a production line for software in which much of the building is automated and humans supervise the line. Agent Scrum treats the retrospective as the place where compounding happens and the sprint loop as the factory line.

3 Setting: the reference implementation

The reference implementation runs on one Linux workstation. A terminal multiplexer (tmux) holds a fleet of persistent AI agents, each a long-running Claude Code session with a defined role: chief of staff, health, finance, communications, research, marketing, customer support, operations, and a chief technology officer (CTO) agent that owns engineering. Each agent has its own message channel to the human owner and can message the others. Scheduled work runs on system timers, not inside the agents.

Engineering work spans several repositories: the fleet's own infrastructure, a consumer mobile and web product, a Rust desktop application, and a machine-monitoring tool. Work is tracked as GitHub and GitLab issues. The developer of record is one person, who acts as stakeholder and final decision-maker but writes little of the code.

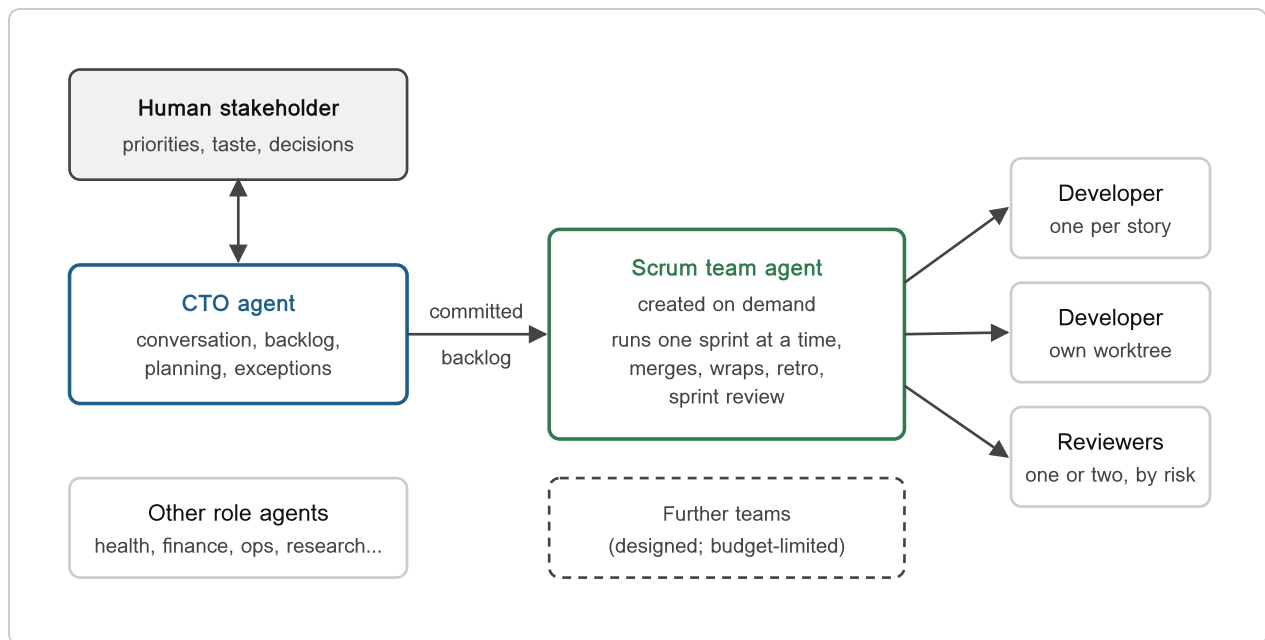


Figure 1. Roles in the reference implementation. The CTO agent talks with the human and plans; the scrum team executes. Dashed: further teams, which the design supports but the token budget does not yet fund.

4 How the method evolved

4.1 Sprints inside the CTO agent (sprints 1 to 234)

The first sprint ran on 20 September 2026. The CTO agent chose a few issues, estimated them in points, started developer agents for each, reviewed and merged their pull requests, and at the end wrote a sprint report with a name, a plan, estimates against actuals, costs, lessons and a retrospective. The report was published as a web page for the human owner to read. In practice it served as the sprint review.

Sprint length settled quickly at two hours, on fixed time boxes that start on odd hours (01:00 to 03:00, 03:00 to 05:00, and so on). As in human Scrum, the length is a choice, not a law (see Sections 7 and 9). By the end of September the loop ran unattended: at the end of each sprint the CTO agent wrote a handoff note, cleared its own context and resumed into the next sprint.

This worked, and it exposed a structural problem. The CTO agent was doing two jobs with one context window. Running a sprint fills the context with build logs, review verdicts and merge state. Talking with the owner about direction, priorities and design needs the same context for a different purpose. Each job degraded the other: sprint state crowded out the conversation, and a conversation in the middle of a sprint delayed merges and wraps.

4.2 Separating conversation from execution

The first design gave the sprint its own permanent agent, provisionally named after Jeff Sutherland, one of Scrum's creators. While designing it, we noticed that nothing limited the design to one such agent. If a scrum master is an agent, a second team is a second instance of the same definition with different repositories assigned to it.

The design that shipped on 19 October therefore defines a *scrum team* as a class of agent, started on demand by the launcher as a numbered instance. The CTO agent keeps the conversation with the owner, the product backlog, sprint planning and exceptions. The scrum team runs exactly what it is handed. A pilot began on 19 October with sprint 235, and since 20 October the dedicated team has run every sprint while the CTO agent runs none. 90 sprints have run that way at the time of writing.

5 The method

5.1 Roles, artifacts and ceremonies

Table 1 maps Scrum's elements to their Agent Scrum equivalents. Most map one to one. The changes are in who plays the roles and in the time scale.

SCRUM	AGENT SCRUM
Stakeholder (sponsor)	The human owner. Sets direction, makes decisions that need judgement or taste, watches the board.
Product owner	The CTO agent. Decides what gets built and why: orders the product backlog, refines it with the stakeholder, plans and commits each sprint backlog, answers the team's impediments, and merges the changes the team may not merge itself (5.5). It is also the architect and the stakeholder's day-to-day engineering conversation partner. It runs no sprints.
Scrum master	The scrum-team agent. Decides how the committed sprint gets executed: starts developers and reviewers, tracks every story's stage, merges what the merge principles allow it (5.5), enforces the deadline, writes the report and runs the retrospective.
Developers	Short-lived developer agents, usually two tasks per story (implement, then a fresh run to finish), each in an isolated git worktree. One or two reviewer agents per story, by risk (5.4).
Sprint backlog	A <i>committed sprint backlog</i> : a file listing each committed story with its points, scope, definition of done, reviewers and a stop rule. It is checked by a linter and committed by the product owner before the sprint starts; the team may not add, resize or reorder stories.
Stretch goals	An Overflow list of ready stories the team may pull in once the commitment is done.
Sprint length	Two hours, on fixed time boxes starting at odd hours, around the clock.
Velocity	Rolling mean of merged points per sprint, used by the linter to reject an over-committed sprint backlog. Paired with token velocity (5.8).
Daily stand-up	None. Replaced by a live board (5.9).
Sprint review	A named sprint report published as a web page at the end of each sprint (5.10).
Retrospective	Part of every wrap. Each finding ends in a filed issue, a mitigation already applied, or a stated reason not to act (5.11).

Table 1. Scrum elements and their Agent Scrum equivalents.

5.2 Planning: from product backlog to sprint backlog

Planning works at four horizons, from months to hours.

- **Issues, epics and milestones (months).** All engineering work is a GitHub or GitLab issue. Larger outcomes are epics, whose body holds a status table of their child issues, and milestones group issues toward a release or goal. Every issue is filed with a priority, a milestone and its epic. A sweep runs before every handoff and after every merge to find work that exists only in a conversation or a note and file it.
- **The product backlog order (weeks).** One short ranked list across all repositories names what comes next. Labels on the issues mirror the ranking, and the planning script reports any drift between the two.
- **The weekly plan (a week).** Once a week the product owner sets two or three goals with finish lines, re-ranks the backlog against them and lays out a ranked queue of stories for the week, sized against rolling velocity. It is reviewed in a 15 to 20 minute backlog refinement session with the human stakeholder, who reorders or cuts; routine stories the stakeholder pre-approves there need no second approval, while changes touching money, authentication or production deploys keep a per-merge approval. The plan also holds a reserve of small one-point stories for sprints that run dry.
- **Sprint planning (hours).** Each sprint backlog is drawn from the weekly queue and committed at least an hour before its sprint starts, and the product owner keeps the next two sprint backlogs committed at all times. Planning revises the weekly plan rather than inventing: new work enters only by displacing a named story or within an interrupt budget of about 20% of the sprint's points, and the report records goal points against interrupt points.

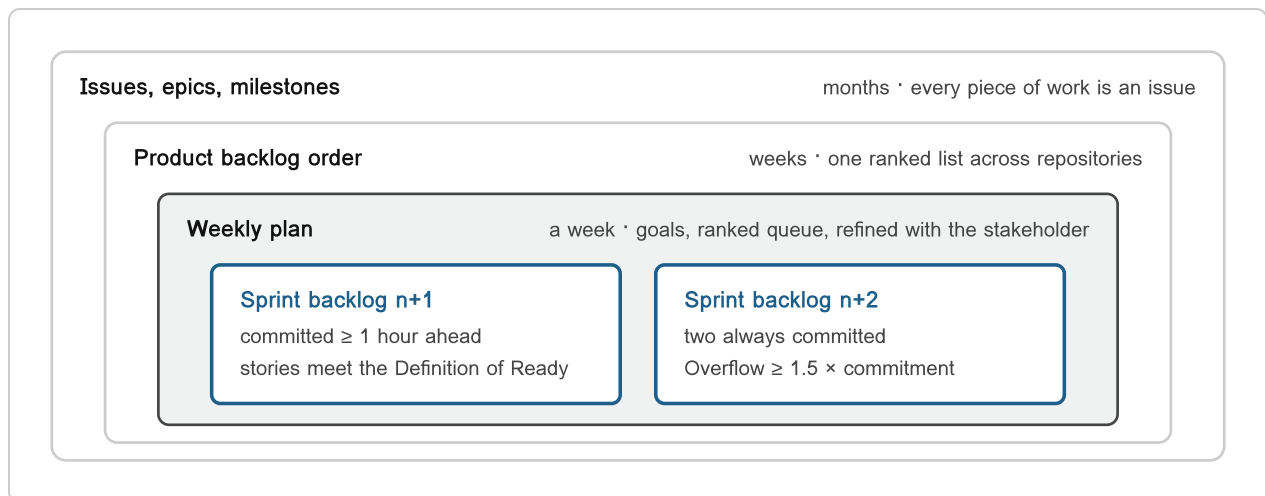


Figure 2. Planning horizons. The product owner works from the outside in; the stakeholder joins at the weekly plan. The scrum team only ever sees a committed sprint backlog.

A story may enter a sprint backlog only when it meets the **Definition of Ready**: the product owner has made four reads of the story, each within 12 hours of its sprint (the issue's log, its open pull requests, its last three comments, and a search of the main branch for an existing fix), sized it, and written a brief and a stop rule. Stories of 5 or 8 points are split or planned first; every sprint also runs one planning-only story that prepares a later one. Each sprint backlog carries ready Overflow worth at least one and a half times its committed points, plus two or three one-point planning stories, so that a sprint whose commitment finishes early still has useful

work. These rules came from measured idle time: on 20 October the team finished its commitment about 40 minutes early in most sprints because nothing ready was left, and on 25 October one sprint sat idle for 45 minutes with no Overflow. The Overflow ratio was raised from one to one and a half on 28 October, after the team ran dry three times in one night.

5.3 The sprint loop

A sprint proceeds as follows. When the sprint starts, the scrum team checks that its sprint backlog is committed and assigned to this team and this sprint, takes a lease on the repositories in scope (so no other team can merge there), names the sprint, records the commitment and starts the first wave of developers. Each developer works in an isolated worktree, opens a pull request and hands it to its reviewers. A blocking review sends the story back for a fix round. Approved changes go through continuous integration and the team's merge script, which re-checks approvals and scope before merging. Ten minutes before the boundary the team stops starting new work and wraps: it saves the state of anything unfinished, merges what is approved, writes the report and the retrospective, writes a handoff note and clears its own context for the next sprint. A story that is approved but not merged at the wrap is not done: it counts as spillover, and its points do not count toward the sprint's done. It is not lost either. Every approval is written to an append-only log, and the team cannot release its lease on a repository while an approved change there is still open. At the next sprint's start the scrum master reconciles that log and sends each approved change back through the merge script, which checks again that the approval is for the current final commit and asks for another review if anything changed. Changes in a class the product owner merges wait in the product owner's merge queue instead.

5.4 Definition of Done

Each story in the committed sprint backlog carries its own done condition: `merged`, `merged+deployed` for the consumer product, where deployment is part of the job, or `closed` with a reason for stories that conclude without code. Underneath that sits a team-wide quality bar that applies to every merge: approval on the final commit by the reviewers the change's risk requires (one for documentation, tests, CI configuration and tested tooling; two for core code, interfaces between components, money, authentication and every consumer-product change), passing continuous integration, and a merge by whoever the merge principles name (5.5). A story that needs a human decision the human has not yet given is not done. It spills over, and the missing decision is recorded as an impediment for the product owner.

5.5 Who merges what

Merging is where a change takes effect, so the method assigns merge authority by principle rather than by repository. Each change is classified from the files it touches, by a script, and each class has exactly one merger.

1. **The team merges what its Definition of Done can fully judge.** An ordinary, reversible change inside the team's leased scope, approved by the reviewers its risk requires and passing CI, is the team's to merge. This covers most work, and keeping it with the team is what lets the loop run unattended.

2. **No agent merges a change to the rules that govern it.** Changes to the agents' own instructions and skills, to the merge rules, to the class lists and to the gates are merged by the product owner. This is separation of duties: a team optimizing its own throughput must not be able to loosen its own constraints, even by accident.
3. **Irreversible or high-consequence changes need the stakeholder's explicit yes.** Money, authentication, production data and schema, public releases, and any weakening of a gate on an irreversible action. The yes is collected at planning and bound to the specific change, so the sprint never waits for it. Where the change is to the gates or merge rules themselves, the product owner merges after the yes.
4. **When in doubt, ask.** If the team cannot tell which class a change belongs to, it raises an impediment instead of guessing (5.7).

The rationale is the same as in human organizations: authority follows the ability to judge consequences. The team can judge whether code is correct. It cannot be the judge of the rules it works under, and only the stakeholder can accept an irreversible risk.

5.6 A hard, deterministic deadline

The time box is the device that makes Scrum work: it forces scope decisions and makes velocity comparable. It is also the hardest device to enforce on agents. An agent cannot be interrupted in the middle of a turn; a message sent to it is read only when its current step ends. An agent's own timers live inside its session and can disappear with no error. In sprint 43 (25 September) the scrum loop's in-session wrap timer never fired, and the agent sat idle for about 100 minutes past its boundary.

The current design separates the deadline from the agent in three layers:

1. **The boundary is data, not a clock reading.** The committed sprint backlog carries the sprint's start and end as timestamps. The team's lease on its repositories expires 60 minutes after the boundary, so a team that overruns cannot hold its scope indefinitely.
2. **An external trigger starts the wrap.** A system timer, independent of the agent's session, fires ten minutes before every boundary. It waits up to ten minutes for the agent to finish its current step, then delivers the wrap instruction. It never interrupts a running step. Of its last 60 firings, 58 delivered and 2 reported an error.
3. **A watchdog catches a missed wrap.** Twenty minutes after the boundary, a second timer checks whether the sprint wrapped. If not, it nudges the agent and then escalates to the operations agent.

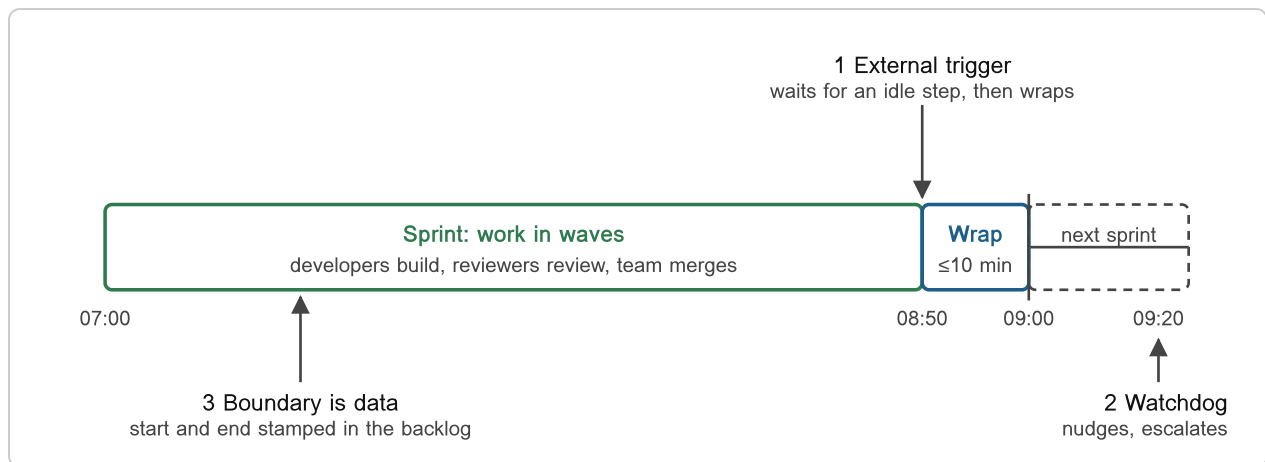


Figure 3. One sprint and the mechanisms that end it. The wrap is requested ten minutes before the boundary by a timer outside the agent, starts when the agent's current step finishes, and is checked again twenty minutes after the boundary.

The deadline is therefore deterministic in when the wrap is requested and bounded in when it starts: up to ten minutes, the length of the longest step an agent may be in. Unfinished stories spill over to the next sprint and are counted by a script at its start.

5.7 Exceptions and impediments

The scrum team never asks the human. It asks the product owner, through an append-only rulings log. A request names its class (scope, merge class, out of work, unclear, blocker), the story it blocks, the question, the evidence, and what the team will do meanwhile; a short message wakes the product owner. The answer is a line in the same log that says whether it applies now or from the next sprint. In its first nine days the dedicated team raised 130 requests. The product owner answered in a median of 1 minute (75th percentile 3 minutes). Most were about scope (82), followed by running out of work (14), merge class (10), unclear rules (8) and blockers (8). Four examples, paraphrased:

- **Scope.** The two reviewers disagreed on a pull request: one blocked, one approved. Fixing the blocking finding needed a file outside the story's declared scope. Ruling, three minutes later: one fix round inside the existing files, with the approach to take.
- **Unclear rules.** The scrum master's instructions said a short re-review counted as approval; the merge script disagreed and refused the merge. Ruling: the script wins and a full review round is owed. The instructions were corrected the same day.
- **Merge class.** A change included an edit to an agent's instructions that had to stay a separate commit, but the team's merge path combines commits. Ruling, one minute later: the product owner merges that change himself.
- **Out of work.** The sprint's stretch work was done and the reserve was empty. Ruling, one minute later: the next sprint backlog is committed, with its source named.

Two exceptions recur most. The first is running out of work. When a sprint's own Overflow runs dry mid-sprint, the team first pulls from the Overflow of the next committed sprint backlog, then from the weekly plan's reserve of one-point stories, and asks the product owner only when both are empty. That request is a single impediment, written to a rulings log the product owner answers; meanwhile the team spends the time on merges, reconciliation and owed reviews. The second is scope. A story's stop rule turns any need outside its declared scope into a question: the team files a scope request with options, and the product owner rules to widen the scope now, to run a fix round, or to spill the story over. Most rulings widen the scope by one file and the story continues. A story added after a sprint backlog is committed is merged by the product owner; the committed backlog itself is never edited. Other impediments in the log include disagreements between reviewers, a change landing in the wrong merge class, and stories that went stale between planning and the sprint. The team never decides scope on its own; scope questions go to the product owner, and questions of taste go to the human.

5.8 Points, velocity and token velocity

Points are assigned by the product owner when it refines a story, on the scale 1, 2, 3, 5, 8; the linter rejects any other value. As in human Scrum they are internal: a point in this project means nothing in another. They become useful through consistency, and the sprint-backlog linter uses rolling velocity to reject an over-committed sprint. Recent sprints commit 13 points. A story's points count in the sprint in which it reached merge approval, even if the merge itself lands in a later sprint; the rule matters less than applying it the same way every time. The figures in this paper predate that rule and count merges.

Human Scrum treats people as the fixed resource. Here the scarce resource is model usage, so each sprint report also records **token velocity**: tokens consumed per merged story point. Recent sprints measured between 0.27 and 0.68 million tokens per merged point (Claude-side only; work delegated to a second vendor's models is excluded). Token velocity is what turns a project's size in points into a budget.

5.9 The board: a sprint radiator

Instead of a daily stand-up, the human watches a live board in a terminal pane: one row of columns per team (Ready, Build, Review, Merging, Done), a card per story with its points and pull request, flags for blocked or silent stories, a burn-up chart against an ideal pace line, and the remaining rate-limit budget. Every stage change a team records becomes an event the board reads. The owner describes it as the change that made the method feel real: work is visible as it flows, and questions can be asked about a specific card while it moves.

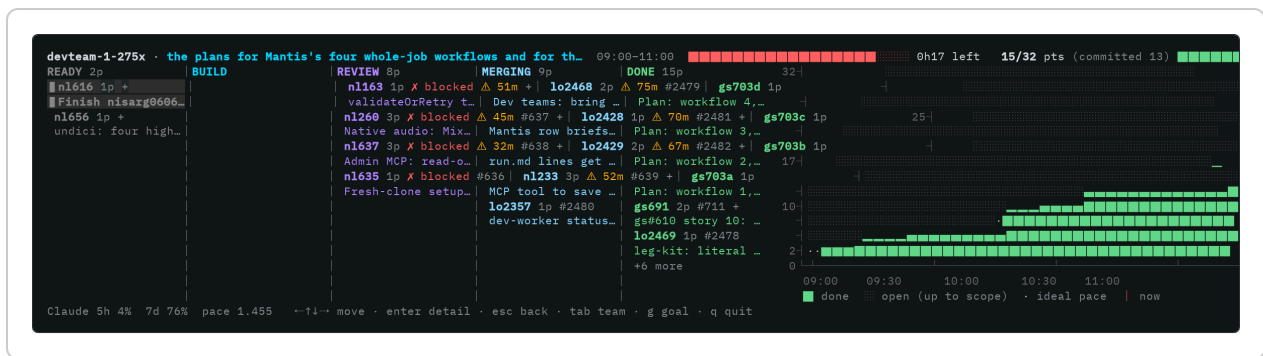


Figure 4. The sprint radiator during sprint devteam-1-275x (09:00 to 11:00), captured from the live terminal at 10:43, with 17 minutes left. Points by column: Ready 6, Review 8, Merging 9, Done 11; the commitment was 13 points and the team had pulled in Overflow. Cards flagged **X** are blocked, and **Δ** marks a story with no event for the minutes shown. The burn-up chart is on the right, and the 5-hour and 7-day rate-limit readings are at the bottom.

5.10 Sprint review

Each sprint is named at planning for what it is meant to achieve (for example "One Build, One Address" or "The File Is Read in the Browser"), and the name is kept in the report. At each wrap the scrum team writes a sprint summary and publishes it as a web page, with its link sent to the stakeholder. The summary serves as the sprint review: the goal and whether it was met, what was merged and why it matters, what was approved but not merged, what spilled over, the impediments raised and how they were ruled, the numbers (commitment against done, points, token velocity) and the retrospective. Appendix A reproduces one.

5.11 Retrospectives as compound engineering

The retrospective is written at every wrap. Each line must end in one of three ways: a filed issue, a mitigation that has already been applied, or a stated reason not to act. Examples from sprint 324: pull requests branched before a timeout fix kept the old limit and timed out (mitigated: merge the fix into the branch and relaunch); the board refused the identifiers of refinement stories, so they never appeared (filed as an issue). This is where the method compounds. A lesson that becomes a rule, a gate or a test changes every later sprint. The unit of compounding is a shipped change to how the next sprint runs; a filed issue only records the opportunity. One complete cycle: in sprint 43 the wrap timer inside the agent's session never fired and the sprint overran by about 100 minutes. The retrospective filed it, the fix moved the trigger outside the agent (5.6), and that trigger has since delivered 58 of its last 60 wraps.

5.12 Human decisions and permission gates

An unattended loop has one dangerous dependency: a point where it waits for a person. Two kinds of wait exist. A *permission gate* is the agent runtime itself pausing to ask whether a command may run. A *question* is an agent asking the human to choose. Either one, raised while the human is away, stops the loop until the human returns, and a stopped agent cannot even run its own timers.

Both have happened. On 22 September a permission prompt in the CTO agent's session halted the sprint cycle for 5 hours 12 minutes. On 26 October the CTO agent opened a question dialog at 15:45 that was answered at 21:53; no sprint backlog was committed in that time, and the scrum team sat idle from 17:00 to 22:00.

The method contains the risk in four ways:

- **Decisions are collected at planning, not during the sprint.** A story that needs the human's approval carries it in the committed sprint backlog as a quoted decision, with the channel and date. Without one, the story is not ready.
- **The scrum team never asks.** A missing decision becomes an impediment for the product owner and the story spills over. The sprint itself never waits on a person.
- **The product owner blocks on a question only under three conditions together.** The human is typing to the product owner right now (recent activity at the terminal is not enough), the answer gates the very next step, and every open question from the team has already been answered, with the next sprint backlog committed. A question that can wait is sent as a message, never as a blocking dialog.
- **Permission prompts are pre-empted.** Agents run with permissions granted in advance, and the gates that remain are hooks that refuse with a reason instead of pausing to ask. Each sprint report counts the permission prompts raised; recent sprints report zero to three, all approved.

The residual risk now sits in planning. The product owner is the one agent that talks to the human, so it is the one agent that can still be held by an unanswered question, and a held product owner commits no sprints.

5.13 Where to start: a minimum viable Agent Scrum

Start with five pieces: a product-owner agent that talks with the human and plans; a separate scrum-master agent that runs one sprint at a time; short-lived developer and reviewer agents; a committed sprint backlog with an explicit done condition for each story; and a wrap trigger that lives outside the agents. Save unfinished work and write a handoff before every context reset, and count a story only when its done condition holds. Add everything else in this section (readiness reads, Overflow ratios, leases, merge classes, the radiator) only as observed failures justify it. That is how each of them arrived here.

What it needs from your platform. The reference implementation runs on persistent agents, but the method does not strictly require them. Its scrum master already clears its context at every wrap and resumes from a handoff note, so each sprint is in effect a fresh session. What the method needs is a scheduler outside the agent that starts each sprint and requests its wrap; state kept in files (the sprint backlog, the question-and-ruling log, the handoff and the sprint summary); and a product owner the human can talk to, which can be an ordinary

interactive coding-agent session. The simplest version is a cron or systemd timer that starts a headless scrum-master session every two hours; that session reads the committed backlog, runs it, and writes its summary and handoff before it exits. Persistent agents make the loop smoother, with a live board and questions answered within a minute, but they are not a prerequisite.

6 Terminology

We use Scrum's vocabulary wherever an equivalent exists, because the method is meant to be recognizable to anyone who has worked in Scrum, and because a shared vocabulary makes the comparison in Section 7 possible. The reference implementation grew its own words first and is being renamed toward Scrum's. One Scrum term shifts slightly: a **task** is still a piece of a story, but here a task is one agent run, so a story is split where an agent session ends rather than by content. Most stories take two tasks (implement, then a fresh run to finish). Table 2 lists the terms with no clean Scrum equivalent. Table 3 connects the method to terms now in use for AI-assisted development, which are newer and less settled than Scrum's.

TERM	MEANING	NEAREST SCRUM IDEA
Seat	One reviewer agent's turn on a pull request. One or two seats, by risk, are part of the Definition of Done.	Code review
Wave	A batch of tasks started together, sized by the remaining rate-limit budget.	None; capacity in Scrum is people
Token velocity	Tokens consumed per merged story point.	Velocity, measured in cost instead of time
Merge tail	Time from a story's approval to its merge, and the stories still approved but unmerged at the end of a sprint.	Part of cycle time (from Kanban)
Scope lease	A team's exclusive right to merge in a set of repositories for one sprint, expiring 60 minutes after the boundary.	None; human teams share a codebase by convention
Overflow	Ready stories the team may pull in after finishing its commitment.	Stretch goals
Handoff and context cycle	At each wrap the agent writes a handoff note, clears its context and resumes from the note.	None; a person carries memory between sprints

Table 2. Terms with no clean Scrum equivalent.

EMERGING TERM	WHERE IT APPEARS IN AGENT SCRUM
Compound engineering	The retrospective. Each finding becomes an issue, a rule, a gate or a test, so later sprints start from a better system (5.11).
Software factory	The sprint loop as a production line: committed work in, merged and reviewed changes out, around the clock, with the human supervising the line instead of working on it.
Orchestrator and workers	A two-level hierarchy. The product owner is the planner: it decides what is built and answers questions. The scrum master is the orchestrator of one sprint: it starts and coordinates short-lived developer and reviewer agents, the workers. Neither does the other's job.
Context engineering	The split between the product owner and the team, which keeps conversation and execution in separate context windows, and the handoff and context cycle at each wrap.
Human on the loop	The stakeholder supervises and decides but is not a step any sprint waits on (5.12).
Agent harness	The deterministic machinery around the agents: the external wrap trigger, leases, the backlog linter and the merge script.
Background or asynchronous agents	Developer agents that run unattended in their own worktrees while the human does something else.
Information radiator	The sprint radiator. The term is from the agile literature (Cockburn); the board is a live one.

Table 3. Emerging terms for AI-assisted development and their place in the method.

7 How Agent Scrum differs from Scrum

- **Time scale.** Two weeks became two hours. Agents do not tire, and the loop runs at night. Short sprints mean fast feedback, small spillover and a velocity sample every two hours. They also mean the fixed cost of planning is paid twelve times a day. The two-hour length is as arbitrary as Scrum's two weeks, and we expect it to be close to the floor: one story takes about half an hour from first build to merge at the median (Table 5) and much longer at the tail, and a sprint has to fit several waves of stories plus its start and wrap. Longer sprints remain an option, with the same trade-off human teams face: less ceremony per point against later discovery of problems.

- **Capacity comes from rate limits, not headcount.** The model provider enforces two usage windows, a rolling 5-hour window and a 7-day window. The 5-hour window caps how many developer agents can work at once; the 7-day window is the team's real capacity for the week. Every committed sprint backlog records a reading of both.
- **No stand-up.** Coordination happens through events on the board, not through a meeting.
- **One estimator.** There is no planning poker. Points come from the product owner alone, which makes them consistent and removes the cross-check that estimation by a group provides.
- **Humans are stakeholders, not developers.** The human makes decisions and approves anything irreversible or customer-facing. A sprint never waits on the human; a story that needs the human spills over.
- **Every role can be multiplied.** A human scrum master is one person. An agent scrum master is a definition that can be started several times.

Why keep sprints at all? An AI engineer's first question is why agents need a time box when an orchestrator could pull ready issues from a queue without end. In our experience the boundary does three jobs that a continuous queue tends to postpone indefinitely. First, and the original motive for sprints here, it forces the agent to write a durable handoff and clear its context on a schedule, before the context fills and quality drops. It also forces unfinished work to be saved and accounted for, so spillover is counted instead of silently accumulating. And it forces an honest comparison of what was committed with what was done, which is what makes velocity, and therefore planning, possible. A queue could do each of these too. The design choice is to make them happen together, on a schedule the agent does not control. Before the first sprint, work in this project ran much like an open queue; the stakeholder asked for sessions that end in a summary and a retrospective, and that request became the first sprint.

8 Observations

All figures are from the reference implementation as of 28 October 2026. They come from one reference implementation and should be read as a case study, not as a controlled evaluation.

MEASURE	VALUE	BASIS
Sprints run	324	20 Sep to 28 Oct 2026; 9 to 12 per day recently
Sprints run by a dedicated team	90	since 19 Oct
Committed stories done in sprint	79%	297 of 375, team sprints
Committed points merged in sprint	71%	310 of 435, 40 sprints reporting both
Median sprint duration, start to wrap	114 min	20 sprints with full event logs
Median start to first work	0.6 min	same 20 sprints
Median last work to wrap	3.9 min	same 20 sprints
Token velocity	0.27M to 0.68M	tokens per merged point, sprint reports 25 to 28 Oct
Retrospective actions filed / closed	159 / 40	issues labelled from a retrospective
Committed points merged in sprint, CTO-run vs team-run	72% vs 71%	114 CTO-run sprints (1 to 131) vs 40 team sprints (280 to 324)
Mean points merged per sprint, CTO-run vs team-run	7.4 vs 7.8	same sprints; committed points only, Overflow excluded
Commits that are reverts	about 1.7%	40 of 2,547 commits across three repositories since 20 Sep
Human stakeholder time	under 2 h/day	the stakeholder's own estimate; usually under 1 hour

Table 4. Summary measures.

STAGE OF A STORY	MEDIAN	75TH PERCENTILE
Build (developer writing the change)	14 min	20 min
Review	2 min	4 min
Fix round after a blocking review	3 min	11 min
Approved to merged (CI and merge)	14 min	19 min

Table 5. Time per stage, from board events for 257 stories. Of these, 130 ended their sprint merged, 55 closed without code, 47 approved but not yet merged, and 15 blocked.

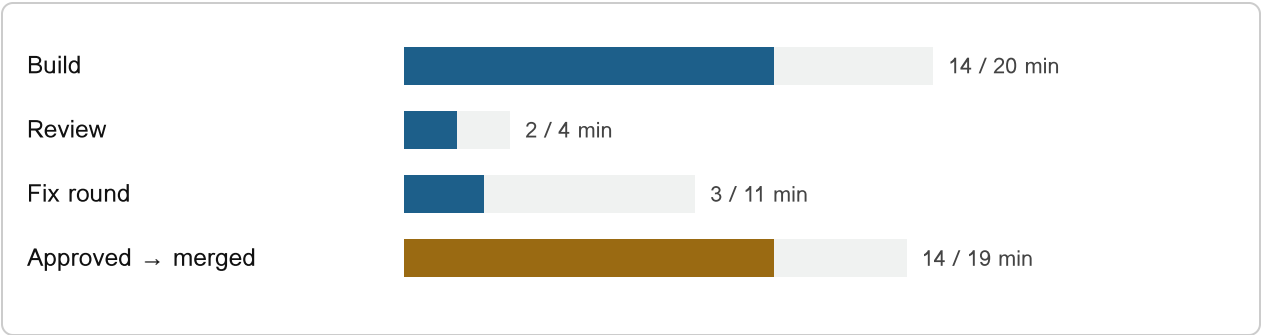


Figure 5. Minutes per stage of a story (dark: median; light: 75th percentile). Approval to merge takes as long as building.

Four observations stand out. **First, the slowest stage is not writing code.** A story's median build takes 14 minutes and its median time from approval to merge also takes 14 minutes, and 18% of stories end their sprint approved but not merged. Continuous integration and merging cost as much time as generation, which makes integration, not more developer agents, the first thing to optimize. **Second, ceremony inside the sprint is small:** under five minutes of a two-hour sprint go to starting and wrapping. Planning happens outside the sprint, in the product owner's session, and has not been measured. **Third, the team usually finishes most of its commitment:** about four stories in five are done inside the sprint, and recent sprints often pull Overflow in after the commitment is met.

Fourth, separating the product owner from the team did not change per-sprint throughput. Committed points merged within the sprint were 72% when the CTO agent ran sprints and 71% under the dedicated team, and points merged per sprint moved from 7.4 to 7.8. The split's value lies elsewhere: the stakeholder's conversation with the product owner no longer competes with sprint state for one context window, and the team became a definition that can be started more than once. That is a claim about the conditions for scaling and about the quality of the human conversation, not about the speed of one team.

Human load. Sprints continue whether or not the stakeholder is present. The stakeholder's time goes into vision and planning conversations with the product owner, usually under an hour a day and never more than about two, with more early in a project or during a change of direction. The recurring human contributions are pushing the work toward user value, documentation and the right priorities. Backlog refinement must happen regularly, and it is efficient because the product owner brings it as a short series of questions.

Delivered work. Process measures say little about what the loop produces, so here are three deliveries, chosen to span the range.

	LIVE SPRINT BOARD (THE RADIATOR)	MS PROJECT POSITIONING IN A DESKTOP APP	DEPLOY RETRY IN A CONSUMER PRODUCT
Asked	The stakeholder, 26 Oct, in a pairing session with the product owner	The stakeholder, 27 Oct at 21:32 and 21:43, in conversation with the product owner	The product owner, 28 Oct at 03:40, after a build service failed four deploys in two hours
Delivered	A terminal board: columns, burn-up chart, story detail, quiet-story warnings, rate-limit readout (Figure 4)	README, user guide, screencast caption and in-app text say the app opens both P6 and MS Project files; a plan for full MS Project parity with a 50,000-task open-time measurement	Deploys that fail with the build service's daemon error retry up to twice; any other failure still fails at once
How	First version built in pair mode in about 40 minutes; hardened by the scrum team in two later sprints	A planning story and a handful of small stories over the three overnight sprints that followed the ask, finished by 05:00	Two sprints: a one-point planning story, then the change with two reviewers and one fix round; about four hours
Human	Paired on the first version; one design ask (events as data, not prose)	The two asks; nothing during the sprints	None
Since	Two open issues and one missing-event bug found while writing this paper	Not yet measured	No defects so far

Table 6. Three deliveries.

Quality. We have not yet measured escaped defects or rework directly. Three mechanisms aim at quality: reviewers scaled to risk (5.4), automated checks in continuous integration, and periodic architecture sprints whose stories pay down structure rather than add features, for example splitting an oversized public interface into modules or adding an architecture check that CI then enforces. As a rough proxy, about 1.7% of commits in the three most active repositories are reverts.

9 Limitations and open problems

- **Retrospective follow-through.** Retrospectives find problems reliably. Fixing them is a different matter: 119 of 159 retrospective issues are open. Compound engineering compounds only when the fixes ship, so retrospective stories need a guaranteed share of each sprint.
- **The merge tail.** Approval to merge takes a median 14 minutes, and 18% of stories end a sprint approved but unmerged. This is the most likely reason throughput sometimes feels slower than the board suggests it should.
- **Unmeasured planning cost.** Refinement and sprint planning happen in the product owner's session and are not yet timed or costed. A full accounting of ceremony needs that number.
- **Human waits in planning.** Sprints no longer wait on the human, but planning still can (5.12). The two sprint backlogs kept committed ahead cover about four hours; a longer wait for an answer still stops the team.
- **No standing backlog refinement.** The product owner refines when it plans. A recurring refinement ceremony that keeps two or three sprints of ready stories would protect teams from running dry.
- **Points are not comparable.** As in human Scrum, points mean nothing across projects. Token velocity partly addresses this, but it varies with the model mix and the kind of work.
- **Several teams at once.** Horizontal scaling is designed but untested.
- **Sprint length is a choice, not a finding.** Two hours has worked, and is probably near the shortest length that works. We have not compared it with longer sprints; we plan to measure ceremony time, spillover, merge tail and token velocity across sprint lengths and move to longer sprints if the numbers favour them.

10 Scaling

The method scales in two directions. **Vertically**, a team can run more developers at once, up to what the 5-hour rate-limit window allows. **Horizontally**, more teams can run at once, each holding a lease on its own repositories. Today a lease covers a whole repository; leases on parts of one repository are designed for but not yet built. The leasing, planning and merge rules already support several teams; what limits the reference implementation to one is the 7-day token budget.

As a snapshot that will date quickly: in October 2026 one dev team at this cadence needed roughly one top-tier Claude Code subscription plus one lower-tier subscription to last through a 7-day usage window. Token velocity makes the next step a budgeting question. If a project's backlog is sized in points, its cost is approximately its points times the token velocity. Adding a project then means adding enough model capacity to run another team, and taking on new work becomes a budget decision rather than a hiring one.

11 Related work

We did not draw on any of the work below; the method grew out of our own failures, one change at a time, and we found this work afterwards. We include it so readers can place Agent Scrum among other approaches. Four lines of work are close to this one. None, as far as we found, combines a time-boxed Scrum loop, a real product backlog worked over weeks, and measurements of the loop itself.

Simulated software teams. Research systems such as ChatDev and MetaGPT assign agents the roles of a software company and pass work through them in a fixed sequence, closer to waterfall than to Scrum [5, 6]; a 2025 literature review describes the field the same way [9]. AgileCoder adds sprints and agile roles (product manager, scrum master, developer, tester) and reports better benchmark results than the waterfall systems [7]. These systems build one program from one prompt in minutes and are evaluated on benchmarks. Agent Scrum works an open-ended backlog across several repositories for weeks, and its results are operational: commitment against done, time per stage, cost per point.

Practitioner arguments that Scrum is breaking. A common 2026 view is that story points and two-week sprints were built to ration scarce human time, so agents make them obsolete; proposals replace points with compute or token budgets and sprints with continuous flow or bounded epics [8]. We agree that the human-paced form breaks. Our experience is that the devices themselves survive once rescaled: the time box shrinks to hours, points stay as a relative measure, and token velocity carries the cost. The bottleneck these authors name, review and validation rather than writing, is also what our stage timings show (Table 5).

Software factories and orchestrators. Systems such as Gas Town and its successor Gas City run tens to a hundred coding agents under a coordinator [10], and the terms *software factory*, *lit factory* and *dark factory* now describe how much human review such a line keeps [11]. That work is mostly about infrastructure: routing, parallelism, health. Agent Scrum is a methodology that could run on such infrastructure. It specifies the cadence, the commitment, the definition of done and the governance of who may merge what. In the factory vocabulary it is a lit factory: humans approve what is irreversible and set direction, and machines verify the rest.

Compound engineering. Every's plan, work, review, compound loop makes each unit of work leave a lesson the system applies next time [12]. Agent Scrum's retrospective is the same idea placed at the sprint boundary, with the rule that every finding ends in an issue, an applied mitigation or a stated reason not to act. Section 9 reports how well that has held.

12 Conclusion

Agent Scrum keeps the parts of Scrum that address coordination (a fixed time box, a committed backlog, relative estimation, a definition of done, a board and a retrospective) and changes the parts that assumed human workers. In this reference implementation the method has run unattended for 324 sprints, finishes about four committed stories in five, and spends almost none of the sprint on ceremony. Its weaknesses are also clear:

retrospective fixes do not ship fast enough, merges lag approvals, and multi-team operation is untested. It seems, at least so far, that Scrum may suit agents as well as, or better than, many of the human teams it was designed for; 38 days is too short to say more. We offer Agent Scrum as a starting point for others organizing teams of agents, and we expect the time scale, the capacity model and the role of the human to be where other implementations differ most.

A Appendix: a sprint summary

The summary of sprint 324, written by the scrum team at its wrap, shortened for length. Story identifiers are the repositories' issue numbers (gs is the desktop application, lo the fleet's own infrastructure, nl the consumer product). "Packet" is the reference implementation's name for the sprint backlog, and a ruling is the product owner's answer to an impediment.

Sprint 324 · One Build, One Address

Team devteam-1 · 05:00 to 07:00 MST, 28 October 2026 · 8 points committed across three repositories

Sprint goal. The docs say `run.sh --local` runs from a detached HEAD and drop the last "nothing uploaded" (gs667); the book clip records the hosted app, the build and address of the README video (gs601); the hosted page's tab says Mantis Gantt (gs669); the story-length strings agree (gs649); the gates job may run 90 minutes (lo2423); the radiator drops its legacy fallback (lo2326); the deploy jobs retry the builder-daemon error (nl663). *Met except nl663, which is approved and in the product owner's merge queue.*

Merged (selected).

- **gs669** (1 pt): the hosted page's title is "Mantis Gantt", pinned by a test assertion that failed on the old title. One reviewer approved.
- **lo2326** (1 pt): the radiator's legacy fallback is gone; a backlog with no events file reads "no sprint packet"; board-window and cursor-edge fixes with tests. Its first CI run timed out at the old 60-minute cap (299 of 322 suites, all passing); the team merged main into the branch, ran one re-review, and it merged at 06:40.
- **gs671** (2 pts, Overflow): the moved-note test helpers live in one module with one press-and-drag; test titles identical before and after.
- **gs678** (1 pt, added by a ruling): the book's clip command builds the hosted app. Merged by the product owner, who merges every story added after commitment.

Approved, not merged. nl663 (2 pts): the deploy job retries a failed build whose output reports a builder-daemon error, at most twice, inside its time budget. The first review blocked on two exit paths that did not report the retry count; one fix commit; both reviewers approved on round 2 (68 tests, 0 failures). Waiting in the product owner's merge queue.

Numbers. Committed 8 points: 6 merged, 2 approved unmerged. Commitment against done: 6 of 7 committed stories merged. Overflow and added stories: 4 points merged, 6 closed as plans or reads. Five impediments raised and answered (one scope, two added stories, one merge class, one out of work). Agents: 16 developer tasks and 2 fix rounds; 23 review reads plus 2 by a second vendor's model. CI: all four runners busy for most of the hour.

Retrospective (selected).

- Pull requests branched before the CI cap was raised kept the 60-minute cap and timed out (three of them) → mitigated: merge main into the branch, one re-review, relaunch; the third is next sprint's first step.
- The merge script refuses stories a ruling adds, and the spillover count reads that ruling as unanswered → filed as an issue.
- The board refused the refinement stories' identifiers, so they never appeared → filed as an issue.
- The scrum master wrote log times from a guessed clock for about 15 minutes → mitigated: a correction line; every later line takes its time from the system clock.
- A review prompt named a command flag that does not exist → mitigated: review prompts name the exact command.

R References

1. Schwaber, K. and Sutherland, J. *The Scrum Guide*. 2020. scrumguides.org
2. Anderson, D. J. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.
3. Cohn, M. *Agile Estimating and Planning*. Prentice Hall, 2005.
4. Cockburn, A. *Agile Software Development*. Addison-Wesley, 2001. (Information radiators.)
5. Qian, C. et al. ChatDev: Communicative Agents for Software Development. ACL 2024. [arXiv:2307.07924](https://arxiv.org/abs/2307.07924)
6. Hong, S. et al. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. ICLR 2024. [arXiv:2308.00352](https://arxiv.org/abs/2308.00352)
7. Nguyen, M. H. et al. AgileCoder: Dynamic Collaborative Agents for Software Development based on Agile Methodology. FORGE 2025. [arXiv:2406.11912](https://arxiv.org/abs/2406.11912)
8. Lee. AI Coding Agents Are Breaking the Two-Week Sprint. HackerNoon, 24 June 2026. hackernoon.com
9. He, J., Treude, C. and Lo, D. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. [arXiv:2404.04834](https://arxiv.org/abs/2404.04834)
10. Yegge, S. Gas City. blog.gascity.com
11. Osmani, A. Software Factories, Light and Dark. 20 July 2026. addyosmani.com
12. Klaassen, K. Compound engineering. Every, 2026. every.to

Created in collaboration with AI.