FACTORY DESIGN

# How a ticket becomes a workflow

Four shapes of AI agent work (sequence, parallel, branch and join, and loop), where a person can step in, and how Widal.Factory runs each one.

SHAPE 1

## Sequence

SHAPE 2

## Parallel

SHAPE 3

## Branch and join

SHAPE 4

## Loop

SUMMARY

# Summary

Widal.Factory takes a ticket that a person wrote in their tracker, such as Linear or Jira, and runs it through six stages: Research, Propose, Plan, Implement, Test and Secure. Each stage is broken into small tasks, so a ticket that would be a day of work for one agent becomes a graph of tasks that each fit in one agent session and that a person can read in a minute. AI agents with different jobs work through the tasks, and people approve the ones that need a person. This paper explains why we model the work this way and not as one long conversation with one agent, and it describes the four shapes the tasks take: tasks in a sequence, tasks that run in parallel up to a set limit, branches that split and join again, and loops that repeat a task until a check passes or a limit is reached. A human gate, which is a task that waits for a named person, can be placed anywhere in any of the four shapes, and the team decides where.

For each shape the paper gives the reason it exists, what the research says about it, what it costs, and how the factory runs it. The last sections cover what a ticket needs to contain so the factory can build the right graph, the roles agents take, and the limits of the approach.

---

TERMS

# The ticket, the stages and the tasks

Five terms are used throughout. A **ticket** is the item a person writes in the tracker. It states the outcome they want and how they will know it is done. The person owns the ticket's title and description, and the factory never rewrites them.

A *stage* is one of the six parts every ticket goes through in the factory's base workflow, from Research to Secure. The next section describes them.

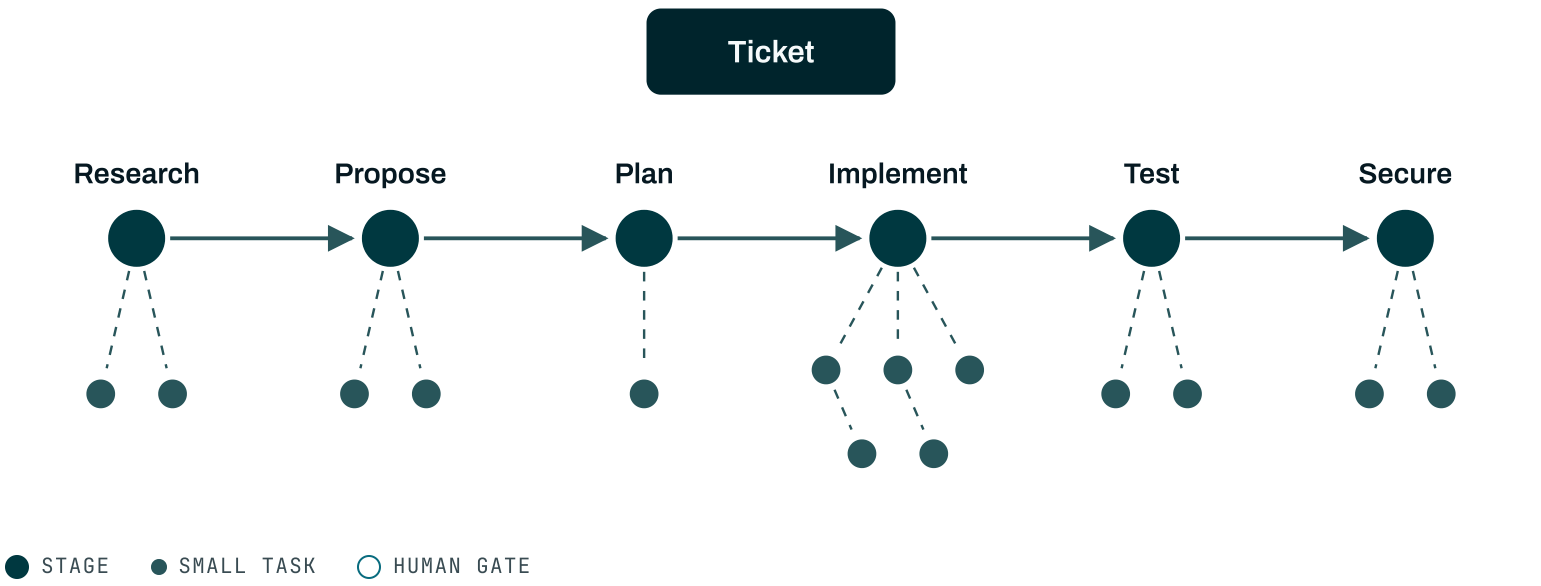
A *task* is a small piece of a stage that one agent session can finish, such as answering one research question, changing one module or running one test suite. A task is open, in progress or closed, and tasks can depend on each other, so the tasks of a ticket form a graph in which a task is ready when nothing it depends on is still open. Tasks are the factory's own records. They are not copied into the tracker as tickets, and the tracker gets one summary comment that the factory keeps up to date.

A *workflow* is a file in the team's git repository that lists the steps for a kind of work, what each step needs before it can start, which agent role takes it, and where a person must approve. Starting a workflow on a ticket creates one task for each step. A workflow is written once and used for many tickets, in the same way a team's

CI configuration is written once and run for every change.

A *human gate* is a task that does no work itself. It blocks the tasks after it until a named person approves, and it records who approved, when and why.

FIGURE 1. ONE TICKET, SIX STAGES, MANY SMALL TASKS




The ticket stays in the tracker and states the outcome. The six stages run in order, and each one is made of small tasks. The number of tasks per stage depends on the ticket: the Plan stage decides how many tasks Implement gets.

WHY A GRAPH

## Why we model agent work as a graph

The simplest way to use a coding agent is to give it the ticket and let it work until it stops. That works for small jobs, but for larger ones it has three problems. A run that takes hours fills the agent's context with earlier work, so later decisions are made with less room for the job at hand. A run that fails halfway leaves no clear point from which to start again. And a person who wants to approve the plan before any code is written, or check a change before it is merged, has no defined place in the run to do so.

Anthropic's guide to building agents draws the same line. It separates *workflows*, where code sets the path and models carry out the steps, from *agents*, where the model chooses its own path, and it recommends workflows when a job can be split into well defined steps because they are easier to predict and to check[1]. The guide names five patterns that recur in practice: prompt chaining (steps in a row), routing (one of several paths), parallelization, orchestrator and workers, and evaluator and optimizer (a loop of attempt and review)[1].

Stripe reports that its internal coding agents, which produce more than 1,300 merged pull requests a week, run inside fixed “blueprints” that mix steps with no model, such as running linters and pushing a branch, with steps where an agent does open-ended work[3].

None of these shapes is new. In 2003, van der Aalst and colleagues catalogued the control-flow patterns found in business process and workflow systems, and the first patterns in their list are the ones this paper uses: sequence, parallel split, synchronization (waiting until all parallel branches are done), exclusive choice, simple merge, and later structured loops[2]. What is new is that most tasks are now done by AI agents, which changes three things. A task costs money in model tokens and not only in compute time. A task can fail in ways a program does not, by doing something plausible but wrong. And many tasks are worth a second look by a person, which a workflow has to make easy without making it mandatory.

A graph of small tasks handles all three. Each task is small enough to run in a fresh agent session with only the context it needs. Each task has its own state, so when a session fails, the task stays open and a new session picks it up from there. The graph shows a person which tasks can run at the same time, which must wait, and where they are asked to decide. It also gives the factory a place to put a limit on spend and on the number of agents running at once.

---

BASE WORKFLOW

## Six stages from ticket to change

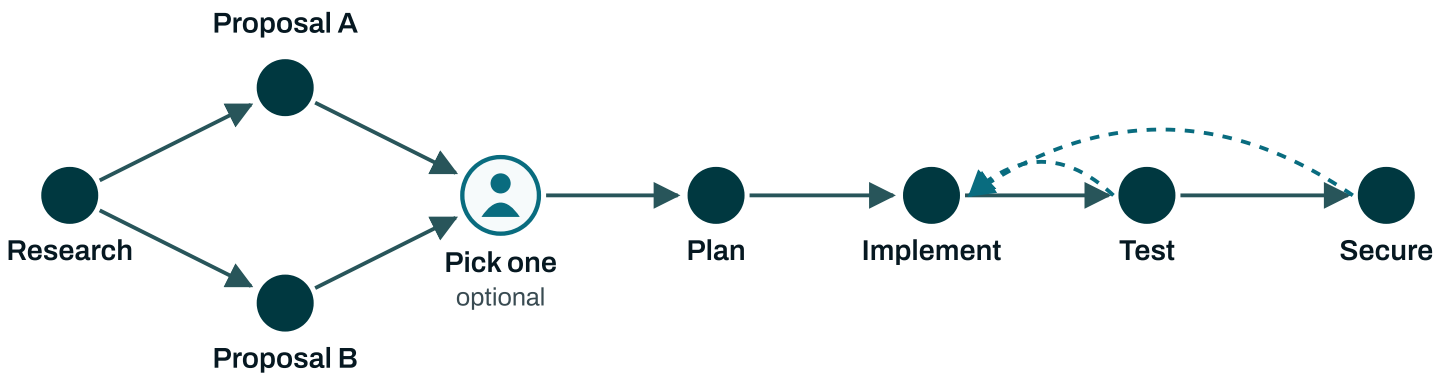
Every ticket in Widal's default workflow goes through the same six stages in the same order. The stages are the outline that does not change from ticket to ticket. What changes is how many tasks each stage has and how those tasks are arranged, which is where the four shapes in the rest of this paper come in.

| STAGE | WHAT HAPPENS | WHAT IT LEAVES BEHIND |
|---|---|---|
| **Research** | An agent reads the ticket, the code it touches, related tickets and the team's documents, and writes down what is known and what is still open. | A research note, and questions for a person if the ticket leaves something open |
| **Propose** | Two agents each write a proposal for how to do the work, independently of each other, with the approach, the trade-offs, the risks and the expected size. | Two proposals, and the one that is chosen |
| **Plan** | An agent turns the chosen proposal into small tasks, with the order between them, which ones can run in parallel, and a check for each. | The tasks of the Implement, Test and Secure stages |
| **Implement** | Builders work through the tasks, each on its own branch or in its own part of the code. | The change, as a pull request |
| **Test** | The team's own tests and the ticket's acceptance criteria run against the change. A failure goes back to Implement with the output. | Test results recorded on the ticket |
| **Secure** | A security review of the change: secrets, new dependencies, changes to permissions and access, how data is handled, and the findings of the team's security scanners. A finding goes back to Implement. | Security findings and how each was resolved |

Two of the stages need a word on why they are there. Propose asks for two proposals because a single proposal tends to become the plan without anyone weighing it, while two independent ones show the choice that is being made. A person can pick between them at a gate, or the Plan stage can pick by rules the workflow states, such as the smaller change or the one that fits the existing code. This is the “voting” form of parallelization in Anthropic's guide, with two votes[1]. Secure is a stage of its own, and not part of Test, because the questions a security review asks are different from whether the code works, and a team often wants a different person, or its security team, to own that gate.

The stages are the default Widal ships, and a team can change them in its own workflow. A small bug fix can skip Propose, a team with its own review process can add a review stage after Secure, and every stage can have a gate in front of it or not. After Secure, the pull request goes into the team's normal review and merge.

FIGURE 2. THE SIX STAGES, WITH THE TWO PROPOSALS




TASK  HUMAN GATE  ---- LOOP BACK

Research feeds two independent proposals. One is picked, here at an optional human gate, and the Plan stage breaks it into tasks. Test and Secure can each send the work back to Implement.

SHAPE 1 **Sequence**

## Tasks in a sequence

The six stages themselves are a sequence, and so are the tasks inside many stages. When the Plan stage breaks a change into tasks that build on each other, such as a database migration, then the code that reads the new column, then the endpoint, then the screen, each task needs the result of the one before it, so they run one after another. Each task depends on the previous one, and each is taken by a fresh session of the agent role the workflow names for it. A researcher that only reads the code can run with fewer permissions than a builder that edits files and pushes a branch.

Splitting a long run into small tasks also makes failures cheaper. When a test fails, the factory sends only the task it belongs to back to Implement, and it does not repeat the research or the planning. Because each task closes with a recorded outcome, such as a commit, a link to the pull request or a test result, the next task starts from evidence and not from the previous session's memory. And because each task is small, a person who opens the ticket's summary can see which part is done and which part is stuck, which a single long run does not show.

FIGURE 3. THE IMPLEMENT STAGE OF ONE TICKET, AS A SEQUENCE OF TASKS




TASK HUMAN GATE

The Plan stage made four tasks for this change, in an order where each one needs the previous one. A gate can sit between any two of them; here none is needed until the change has passed Test and Secure.

The cost of a sequence is its length. The time a ticket takes is the sum of its tasks, plus the time each gate waits for a person. The wait at a gate can be longer than the agent work around it, so where a team puts its gates can matter more for speed than how fast the agents are. A sequence is the right default because it is easy to read and to debug. The other shapes are worth their extra complexity only when the work has the structure they need.

SHAPE 2 **Parallel**

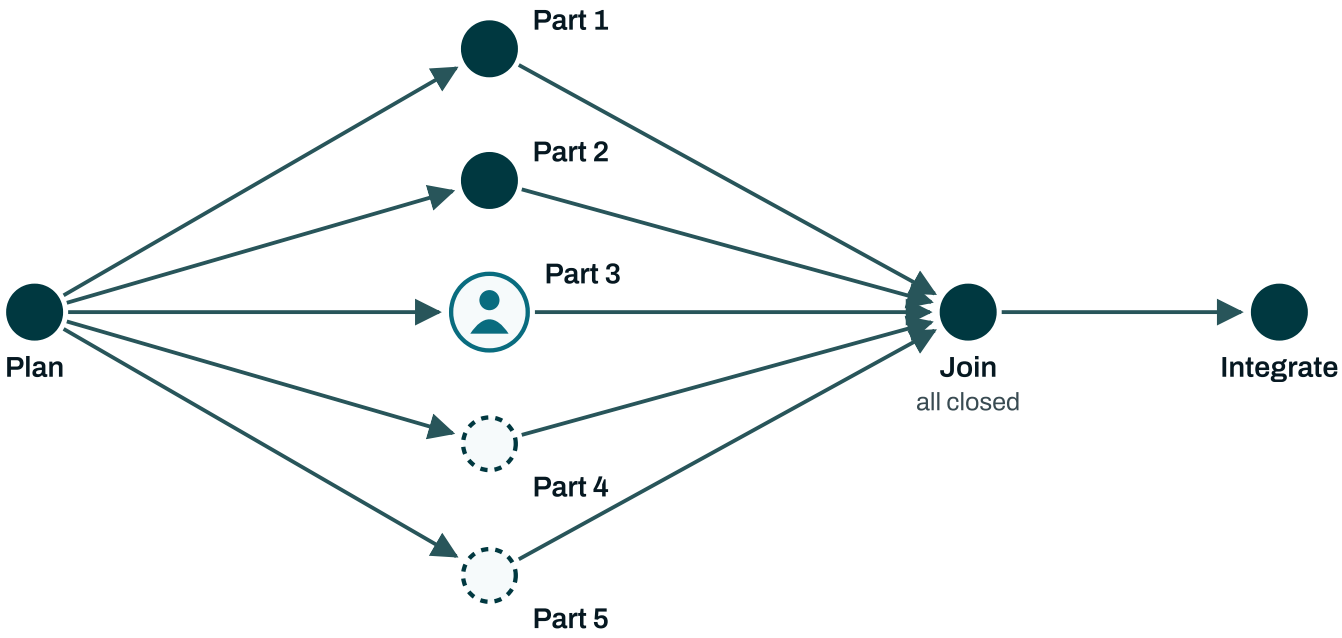
## Tasks that run at the same time, under a limit

Some work splits into parts that do not depend on each other: the same change in five services, a set of failing tests in different modules, a dependency upgrade in several repositories, or research questions that can be answered separately. Those parts can run at the same time, and a later task waits until all of them are done before it starts. Van der Aalst calls this a parallel split followed by synchronization[2].

The evidence on parallel agents points both ways, and the difference is whether the parallel parts share decisions. Anthropic reports that its research system, in which a lead agent starts several subagents that search in parallel, scored 90.2% better than a single agent on Anthropic's internal research evaluation, and that it used about 15 times as many tokens as a chat[4]. Anthropic adds that such systems pay off only when the task is valuable enough to cover the cost, and that tasks in which all agents need the same context, or that have many dependencies between agents, fit less well, and that most coding tasks have fewer parts that can truly run in parallel than research does[4]. Cognition, the company behind the Devin coding agent, argued in June 2025 that subagents working in parallel on parts of one product make choices that conflict, because each one sees only its own part of the earlier decisions[5].

Our reading is that parallel tasks work when each one has its own files, its own repository or its own question, and when the decisions they would share are made before the split, in the Plan stage, and written into each task's description. When the parts would edit the same code, the factory runs them in a sequence instead. The Plan stage makes this explicit: it lists which tasks are independent, and only those run in parallel. The Research stage is often the easiest place for parallel tasks, because separate research questions share no code at all.

FIGURE 4. FIVE PARALLEL TASKS WITH AT MOST THREE RUNNING




The plan splits the work into five independent parts. Parts 1 and 2 run, part 3 waits for a person, and with a limit of three running tasks, parts 4 and 5 wait for a slot. The join waits until all five are closed, and the person deciding on part 3 does not hold up parts 1 and 2.

## The limit on concurrency

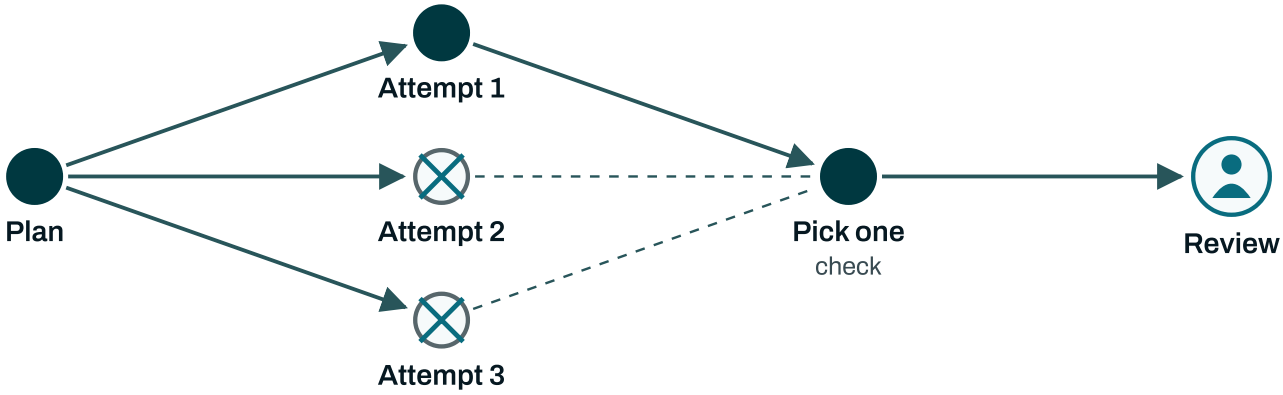
Every parallel shape needs a limit on how many tasks run at once. Without one, a plan that lists forty parts starts forty agent sessions, which spends forty times the tokens in the same hour, can hit the model provider's rate limits, and produces forty changes for people to review at once. The factory sets the limit per agent role, so a team can allow three builders and ten readers at the same time. Tasks over the limit stay ready and start as soon as a running task closes, which is the dashed state in Figure 4.

A limit also sets the expected speed-up. The time a parallel workflow takes is bounded by its longest chain of dependent tasks, however many agents run beside it, which is the point Amdahl made about parallel computers in 1967[6]. Five parallel parts behind the Plan stage and in front of an integration task save the time of four parts, at best, and nothing on the plan or the integration. A workflow author should look at that chain before adding agents.

### Several attempts at the same task

A special case of the parallel shape is to run the same task several times, independently, and keep the best result. The Propose stage does this with two proposals on every ticket. This is the “voting” form of parallelization in Anthropic's guide[1]. The same idea can be used later: for a hard ticket the factory can start, for example, three builders on the same plan, run the tests on all three branches, and send the one that passes with the smallest change to review, closing the other two without merging them. The cost is easy to state in advance: three attempts cost about three times one attempt, so a team that caps a ticket at a certain spend can afford fewer attempts on expensive tickets and more on cheap ones.

FIGURE 5. THREE ATTEMPTS, ONE KEPT




Each attempt is a separate task on a separate branch. A check task picks one by a rule the workflow states, for example all tests pass and the smallest diff, and the others are closed with the reason recorded.

SHAPE 3

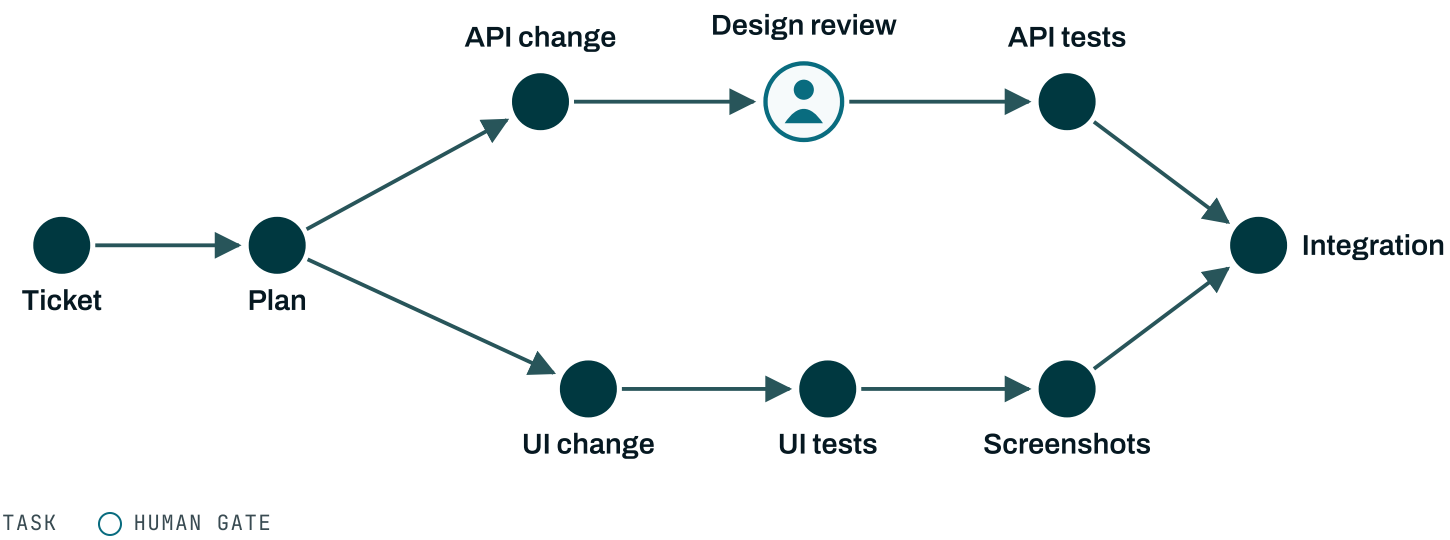
## Branch and join

# Branches that split and join again

A branch is a point where the work takes more than one path before it comes together again. There are two kinds, and a workflow should say which one it means. In a parallel split, both paths run: a change that needs a database migration and an update to the user interface can have one branch for each, with different agent roles and different checks, joined by an integration test that needs both. In an exclusive choice, only one path runs, chosen from data that is available when the work reaches the branch: a ticket labelled as a bug may go through a task that first reproduces the bug in a failing test, while a ticket for a new feature goes through a design task instead[2]. Anthropic's guide calls the second kind routing[1].

The two branches do not need the same controls. In Figure 6 the branch that changes the API has a gate where a person approves the design of the new interface, while the branch that changes the screens runs its tests without a person. The join waits for both, so the integration test never runs against half a change.

FIGURE 6. TWO BRANCHES WITH DIFFERENT CONTROLS




The split is a parallel split: both branches run. The upper branch changes a public interface, so the team has put a design gate on it. The lower branch has only automated checks.

The factory handles the two kinds differently. A choice that can be made when the workflow starts, such as the ticket's type or a label, is made then: the tasks of the path that does not apply are left out of the graph, so nobody sees tasks that will never run. A choice that depends on the result of an earlier task, such as whether the plan found that a migration is needed, is made by a decision task. Its agent records the result, and the factory routes the next task to the branch the result names. When the choice carries real risk, the decision task can be a human gate, so a person picks the path.

SHAPE 4

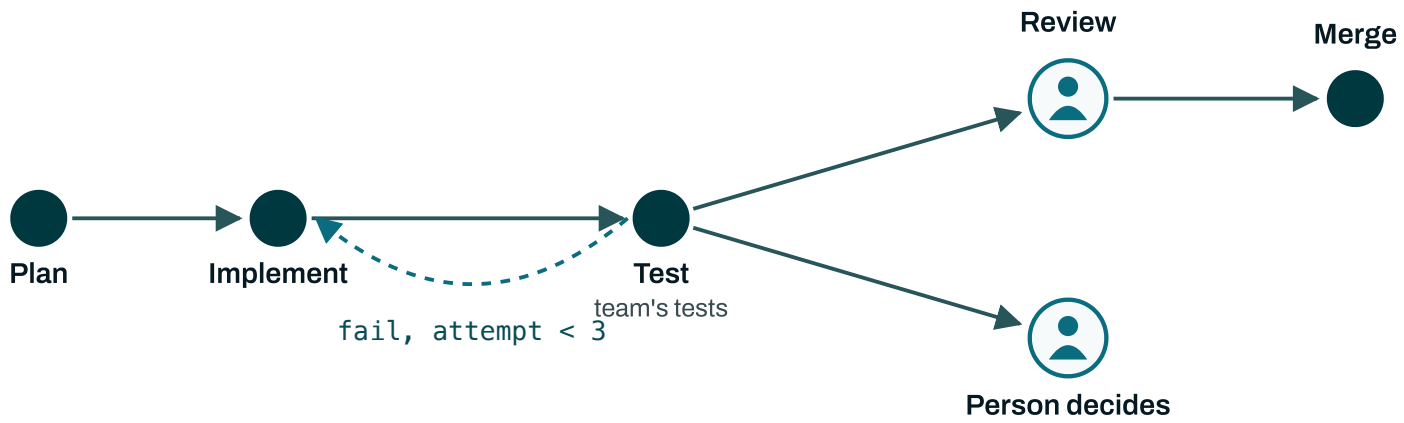
## Loop

# Loops with a limit

The fourth shape repeats a task. The most common loop in agent work is build and check: an agent makes a change, a check runs, and if the check fails, the agent gets the failure and tries again. Anthropic's guide calls this the evaluator and optimizer pattern[1], and the Self-Refine study found that letting a model critique and revise its own output improved results by about 20 percentage points on average across seven tasks, compared with a single attempt[7]. In the base workflow, both Test and Secure close such a loop: a failing test or a security finding sends the task it belongs to back to Implement. A review loop has the same shape on a longer timescale: a reviewer, who can be a person or an agent, leaves comments, and a fresh session addresses them on the same branch.

Every loop in the factory has a limit, and the limit is written in the workflow. A loop can be limited by a number of attempts, by spend, or by time, and when it reaches the limit it stops and hands the work to a person with the history of each attempt attached. Without a limit, an agent that cannot make a failing test pass will keep trying, and the cost grows with each attempt while the chance of success falls. The check that decides whether to loop should be a program where possible, such as the team's test suite or linter, because a program gives the same answer every time and an agent may not.

FIGURE 7. AN IMPLEMENT AND TEST LOOP THAT HANDS OVER TO A PERSON




UPPER PATH: CHECK PASSES

LOWER PATH: THIRD FAILURE, A PERSON DECIDES

● TASK ○ HUMAN GATE ---- LOOP BACK

The check is the team's own test command. A failure sends the work back to the Implement task with the test output. After three failed attempts the loop stops and a person decides what happens next.

Two other loops are common. A retry loop runs a task again when it failed for a reason that is likely to pass on a second try, such as a network error or a check that timed out, and it does not count those failures the same way as a wrong answer. A loop over a list runs the same tasks once for each item, such as each file that needs a migration or each customer configuration that needs an update. A loop over a list is a sequence when the items must be done in order and a parallel shape when they need not be, and it is subject to the same limit on concurrency.

PEOPLE

## Where a person steps in

A human gate can sit anywhere in the four shapes: before the build, on one branch, inside a loop after a number of failed attempts, or before the merge. The factory treats a gate as a task like any other, which means the team's workflow, and not the factory, decides how many gates there are and where. Different teams want different answers. A team that is new to agents may put a person on the plan, the design and the merge, and remove gates as it learns where the agents are reliable. A team with strong tests may keep only the merge gate. Automatic approval of a gate is an option a workflow can turn on, and it is off in the workflows Widal ships.

Established workflow tools treat a human decision in the same way, as a task that waits. Temporal's approval pattern blocks a workflow until a signal with the decision arrives, with a timeout, and records the approver, the reason and the time[8]. GitHub Actions lets a repository require that up to six named people or teams approve a job before it deploys, and can stop the person who started a deployment from approving it[9]. LangGraph, a library for agent graphs, saves the graph's state when a task asks a person and resumes from that point with the answer[10].

A gate in the factory has the same properties:

**01** It names who may decide: a person, a role such as the ticket's assignee, or a team.

**02** It tells that person where they already work, such as Slack or the factory's hub, with what is being asked and the evidence for it, and it reminds them if nobody answers.

**03** It can have a timeout and states what happens when the timeout passes.

**04** It records who decided, when, and the reason, on the task, so the decision is part of the ticket's history.

05 It lets the person answer more than yes or no. A rejection with a comment sends the work back into a loop, so the gate becomes the reviewer in the review loop of Shape 4.

An agent can also stop and ask a person a question in the middle of a task, for example when the ticket leaves a choice open. That question is not a gate in the workflow: the task stays in progress and the session waits for the answer. If the same question comes up often, it is a sign that the ticket template should ask for that information, or that the workflow needs a gate at that point.

INPUT

## What a ticket needs so the factory can build the graph

The graph is only as good as the ticket it comes from. A ticket that says “fix the login” can only become a single task, because nothing in it says what the parts are or how anyone will know the work is done. The table lists what we ask a team to put in a ticket, and what each item becomes in the graph. Most teams already write some of this, and a ticket template in the tracker can ask for the rest.

| IN THE TICKET | WHAT IT BECOMES IN THE GRAPH |
|---|---|
| **The outcome, in one or two sentences** | The goal every task's description starts from |
| **Acceptance criteria that a test or a person can check** | The final check task, and the evidence it records before the ticket can close |
| **The areas of code, services or repositories involved** | The parts of a parallel shape, or the branches of a split |
| **Order between parts, if any** | Dependencies between tasks, which turn parallel parts into a sequence |
| **The kind of work, such as bug, feature or upgrade** | The workflow that is used, and the path taken at a choice known at the start |
| **Who must approve what** | The human gates and the people they name |
| **A budget or a priority** | The limits on attempts, spend and concurrency |

The first three stages fill gaps. When a ticket names the outcome and the acceptance criteria but not the parts, Research finds the code involved, Propose sets out two ways to do it, and Plan writes the parts and their order as tasks. A team that wants to see the plan before any code is written puts a gate after the Plan stage. Over time, the plans that people approve without changes show which kinds of ticket can skip that gate.

AGENTS

## Agents with different jobs

Each task names the agent role that takes it, and each role has its own instructions, tools and permissions. Splitting the work by role keeps each session focused, and it lets a team give each role only the access its tasks need. The roles below follow the six stages, and they are an example of how a team might divide the work. A team can rename them, merge them or add its own.

| ROLE | WHAT IT DOES | WHAT IT MAY TOUCH |
|---|---|---|
| **Researcher** | Research: reads the ticket, the code and the documents, and writes what is known and open | Reads the repository and documents |
| **Planner** | Propose and Plan: writes a proposal, or turns the chosen one into tasks | Reads the repository; writes tasks |
| **Builder** | Implement: makes the change on its own branch | Edits files in its own copy of the repository; pushes its branch |
| **Tester** | Test: runs the team's checks and reports the result | Runs the test commands; cannot edit the change |
| **Security reviewer** | Secure: reviews the change for security problems and reads the scanners' findings | Reads the change and scan results; comments on the pull request |
| **Auditor** | Checks that the run followed the workflow and that the evidence is recorded | Reads tasks and logs; changes nothing |

The roles also make guards easier to place. A command that deletes data or rewrites history can be blocked for every role by default and allowed only where a task needs it, and a list of allowed commands per role is shorter and easier to review than one list for an agent that does everything. The factory runs each session in its own isolated machine with only the credentials of the person it works for, so a mistake in one task cannot reach the work of another.

ENGINE

## How the factory runs each shape

Widal.Factory runs workflows on an open source workflow engine that keeps every workflow, task and gate as a record in a shared database. The engine starts an agent session for each ready task, and each session takes one task, works it, records the outcome and ends, so a crashed session loses no work: the task stays where it was and a new session picks it up. The table shows how each shape is expressed.

| SHAPE | HOW A WORKFLOW EXPRESSES IT |
|---|---|
| **Sequence** | Each task lists the tasks it needs. A task becomes ready when those are closed. |
| **Parallel** | Tasks that do not need each other are ready together. A task can also fan out over a list, starting the same tasks once per item, at once or one at a time. Each agent role has a maximum number of sessions, which is the limit on concurrency. |
| **Join** | A task that needs all the parallel tasks. The workflow's final task combines their outcomes into one result. |
| **Choice** | A choice known at the start is a condition on a task, decided when the workflow starts. A choice made later is a decision task that routes the next task. |
| **Loop** | A loop over a count or a list is expanded when the workflow starts. A check loop runs a program after each attempt and starts another attempt until the program passes or the limit of attempts is reached. A retry loop repeats tasks that failed for a passing reason. |
| **Human gate** | A gate task blocks the tasks after it until it is closed. The factory's hub closes it when the named person decides, and records the decision. |
| **Failure** | Tasks can be grouped so that when one fails the rest of the group is skipped, while clean-up tasks still run. |

Workflows live in the team's git repository and change through pull requests, like the rest of the team's code, so a change to where the gates are is reviewed the same way as a change to the tests. The factory's own default workflows are written in the same format, and a team can extend them or replace single steps without copying the whole file.

LIMITS

## Limits and open questions

The shapes in this paper describe how we design workflows, and they have limits that a team should know before it relies on them.

**01** Loops that repeat until a condition written in the workflow becomes true are accepted by the engine but not yet carried out by it, so today a loop is either a fixed count or list, or a check loop driven by a program.

**02** A join that continues when any one of several branches is done, instead of all of them, is not yet interpreted by the engine. Workflows that need it use a decision task.

**03** Parallel agents that edit the same code still conflict. The Plan stage reduces this, but it depends on the planner splitting the work well, and we do not yet have numbers on how often a split has to be redone.

**04** We do not yet have published measurements of how much each shape changes the time or the cost of a ticket in a real team. The figures quoted from Anthropic and Self-Refine come from their own tasks, which are not ticket work.

ABOUT

## About Widal

Widal, Inc. builds Widal.AS, one agent system for a whole company, with Widal.Factory as its part for engineering teams. Widal's engineers install and run the system with each client, in the client's own cloud account. Contact: hello@widal.com.

METHOD

## How this paper was compiled

The four shapes come from Widal's factory design and from the workflow sketches our team drew for it in November 2026. The description of the engine was checked against its specification and against Widal.Factory's design documents on 22 November 2026. The outside evidence comes from the sources below, read on or before that date. Several are reports by companies about their own systems, so their figures show what those companies observed and are not independent measurements.

# Sources


1. Anthropic, "Building effective agents", 19 Dec 2024. www.anthropic.com/engineering/building-effective-agents
2. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski and A. P. Barros, "Workflow Patterns", Distributed and Parallel Databases 14(1), 2003, with the later catalogue on the Workflow Patterns site. doi.org/10.1023/A:1022883727209 and www.workflowpatterns.com
3. Stripe, "Minions: Stripe's one-shot, end-to-end coding agents, part 2", 19 Feb 2026. stripe.dev/blog/minions-stripes-one-shot-end-to-end-coding-agents-part-2
4. Anthropic, "How we built our multi-agent research system", 13 Jun 2025. www.anthropic.com/engineering/multi-agent-research-system
5. W. Yan, Cognition, "Don't Build Multi-Agents", 12 Jun 2025. cognition.ai/blog/dont-build-multi-agents
6. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", AFIPS Spring Joint Computer Conference, 1967. doi.org/10.1145/1465482.1465560
7. A. Madaan and others, "Self-Refine: Iterative Refinement with Self-Feedback", NeurIPS 2023 (arXiv 2303.17651). arxiv.org/abs/2303.17651
8. Temporal, "Approval pattern", documentation, read Nov 2026. docs.temporal.io/design-patterns/approval
9. GitHub, "Deployments and environments", GitHub Actions documentation, read Nov 2026. docs.github.com/en/actions/reference/workflows-and-actions/deployments-and-environments
10. LangChain, "Interrupts", LangGraph documentation, read Nov 2026. docs.langchain.com/oss/python/langgraph/interrupts