Memex AI

Understanding Memex

Why we built it, the four principles it runs on, the one idea everything rests on, the features that set it apart, and how to begin.

Welcome.

This document exists to do one thing: leave you understanding Memex well enough to decide whether it is for you, and to use it with intent rather than by trial and error once you are in. It is not a click-by-click tutorial. It is the thinking behind the system, so that when you meet a decision record or an acceptance criterion or a readiness check, you already know why it is there and what it is for.

Read it once, start to finish. It takes about ten minutes, whether you are weighing Memex up or already setting up your first spec, and it will save you a great deal more than that.

It comes in two parts, with a short FAQ at the close.

Part One: The Thinking Behind Memex is why it exists, what it believes, and how it works.

Part Two: Getting Started is the path to your first spec. Read straight through if you are weighing Memex up; if you have already decided, jump to Part Two and begin.

Part One - The Thinking Behind Memex

Why Memex exists

For most of the history of software, a specification was a document. A Word file, a design doc, an RFC, a ticket with a PDF attached, a wiki page with twenty comments down the side. It was written, half-read, and quietly abandoned the moment the first line of code landed. That was tolerable, because the engineer who wrote the code had sat in the planning meeting and could fill the gaps from memory. The document did not have to carry the whole truth, because a human did.

That arrangement has broken. On an AI-native team the participant writing the code is increasingly an agent, and the agent did not sit in the meeting. It has only the artefact you hand it. If that artefact is a long piece of prose, the agent loads all of it, parses it, and acts on whichever part it noticed first. It fills the gaps you left with confident invention. The result is fluent code that does precisely the wrong thing, produced very fast.

The instinct is to write a better document. A tidier one, a more rigorous one. That instinct is wrong. The problem is not the quality of the prose. It is the shape of the artefact. A document cannot tell you it has been verified. It cannot raise a problem back to you. It cannot know it has been overtaken by a change somewhere else. To run a team where agents do most of the building, the specification has to stop being a document and become something an agent can query, write back to, and be held to.

That something is a database. Memex is the system that makes the specification one.

Memex versus a ticket tracker and a Markdown spec

If you are deciding whether Memex earns a place in your stack, the fair question is what it gives you that the two tools you already use do not. Almost every team manages work in one of two ways: a ticket tracker like Jira, or specifications written as Markdown or Word documents. Neither is a bad tool. Both were built for a world where a person carried the context in their head and the artefact only had to jog their memory.

That is the assumption that breaks when an agent does the building, because the agent has no memory to jog. It has only the artefact.

Here is where that leaves each one.

	Jira / ticket tracker	Markdown / Word spec	Memex
A unit of work is	a flat ticket	a section of prose	six linked records
Decisions live	in comments nobody rereads	buried in the prose	first-class records
"Done" means	a status someone drags	nothing; prose can't self-report	derived from the tests
An agent can	read text, no scoped context	load the whole file	query the exact slice
When reality moves on	the ticket silently rots	the document silently rots	drift detection flags it
Memory across work	none; each ticket an island	none; starts from scratch	a knowledge graph

A ticket and a document are both *passive*: they sit there, and an agent cannot safely act on either without a human filling the gaps from memory. Memex makes the specification *active*, a live surface your team and your agents read from and write to together, holding itself to a standard the document never could.

If your team is still mostly humans reading tickets, Jira works and you may not feel the gap yet. The moment agents start doing real building, the gap is the difference between code that does what you meant and code that does what the artefact happened to say.

What we believe

Four principles sit underneath everything Memex does. They are not features. They are the convictions that decide what gets built and what does not.

AI is the operating system, not a tool. In most companies AI sits beside the work: an assistant you open, take a suggestion from, and close. The work itself still runs on the old rails. We believe the relationship has to invert. The work should run *through* the intelligent layer, which means every decision, every check and every unit of work has to live somewhere an agent can read and act on.

Every important process is a closed loop. Most teams run open loops: a decision is made, executed, and never checked against its outcome. A closed loop watches its own output and corrects. We believe the act of building software should be wrapped in one, so that "done" is something the system observes rather than something a person asserts.

The work must be machine-readable. An agent can only act on what you can show it. If your decisions live in someone's head and your rationale lives in a Slack thread, you have nothing to hand an agent but folklore. We believe every important action should leave a structured artefact the system can learn from. The test is simple: give the models as much context as you would give a new employee.

Engineering runs as a software factory. The human writes the specification and the tests that define success. The agents generate the implementation and iterate until the tests pass. The human defines what to build and judges the result; the code itself is the agent's job. We believe this is the next turn of test-driven development, and that teams who reach it work at a cadence the old shape of the work could never sustain. This is the boldest of the four principles and the least settled; we hold it as where engineering is heading, not as something already true of every team, and the rest of Memex works whether you take it all the way or not.

Everything below is a consequence of these four. If you ever wonder why Memex makes you do something a simpler tool would let you skip, the answer is almost always one of these principles refusing to bend.

The one idea: a spec is a database, not a document

This is the core mental model, and once it lands the rest of the system explains itself.

A document is prose with sections and headings. A database is records with types, statuses and relationships, where a query returns exactly the slice the caller needs, whether the caller is you reading in a browser or an agent loading the decisions that constrain the task it just claimed.

In Memex, a specification is six kinds of record, each a first-class object with its own status and history.

- **The narrative.** What is being built and why. Ordered prose covering overview, design, architecture and risks. This is the part closest to the old spec, but now it is one record among six rather than the whole thing.
- **The decisions.** Every non-obvious choice the build depends on, written as its own addressable record with its options, its rationale, and the path chosen. When an agent meets a decision the spec did not anticipate, it does not invent. It writes a *candidate decision* for you to review.
- **The acceptance criteria.** Observable, testable conditions that define what "done" means. *A returning user with an expired session sees the sign-in page, not an error.* Each has a live status, and the status is not self-reported. It is derived from your tests.
- **The tasks.** A dependency graph of the work, not a flat backlog. Each task carries its goal, the criteria it satisfies, and its dependencies. An agent asks which tasks are ready and gets back only the ones with no open blockers.
- **The issues.** When an agent hits something it cannot resolve, it writes an issue record against the spec rather than burying the problem in a transcript nobody reads. The issue surfaces where you can see it, and can be promoted into a task with a verifying criterion attached.
- **The emit tokens.** The signal that keeps each criterion's status honest. A small tag in a test reports its result back to the criterion it covers, so a status is never self-declared.

Those six records, joined to one another in one place, are what a specification becomes in Memex. Every decision links to the criteria it spawned. Every criterion links to the tasks that satisfy it and the tests that verify it. Every issue links to the task that surfaced it. The artefact is no longer prose anyone can ignore. It is a graph you and your agents read and write together, and it is your unit of work: you will spend most of your time inside a single spec.

The features that set it apart

The six records are the shape of the system. Three features are what the shape buys you, and they are why Memex is more than a tidier place to keep documents.

The knowledge graph. Every spec, decision and standard you create feeds a single memory the whole team draws on. It holds two things. Your *standards*, the rules and conventions you want every build to respect, kept current under your control and referenced continuously rather than forgotten in a style guide. And your *history*, every spec, acceptance criterion and resolved decision, indexed so the relevant slice of past work is there before the next piece starts.

What makes this memory more than a store is a single call your agents make and you never do: `search_memex`. Every coding session begins with it. Before an agent writes a line, it runs a fast semantic search across your standards, specs, criteria and decisions, and loads what the team has already settled. No session starts blind: rather than ask what the team decided about authentication last quarter, the agent retrieves that decision, with its rationale, and builds on it. And because the graph is spec-aware and person-aware, it knows which spec relates to which, which decisions a piece of work inherits, and who holds the knowledge behind it. An agent on its first task builds on the team's prior decisions instead of guessing. Context stops leaking; the team's history finally works for the team.

Drift detection. This is the feature that makes Memex something a wiki can never be. A document goes stale silently; nobody is told when reality has moved past it. Memex watches for that divergence on four fronts: an agent can report drift while planning a piece of work, a changed decision can trigger it, a completed task can trigger it when the implementation

no longer matches what was agreed, and anything flagged lands in a *Drift Inbox* with a proposed correction to the code or the rule. Drift stops being something you discover months later in an incident and becomes something the system raises to you while it is still cheap to fix.

Decision extraction. Nobody logs decisions by hand. Every team that has tried this has watched the practice die within a fortnight. So Memex never asks you to. It extracts decisions as *candidates*: the implicit choices hiding inside a narrative are surfaced before build, the choices an agent makes mid-build are proposed back to you to approve or reject, and each candidate carries its options and rationale through a short review before it joins the decision graph. The record builds itself as a by-product of the work, which is the only way a decision log ever survives.

What it looks like in practice

None of this is abstract, and the clearest way to feel it is to watch it across two specs at once. A single large piece of work, broken into two specs whose decisions interlock, is exactly the case that used to live in one person's head and fall apart the moment they stepped away. In Memex it happens in the open, because the specs know about each other.

Two specs reach the same point, one a prerequisite of the other, and Memex holds the relationship between them:

```
Two specs now sit ready in specify, both one step from build:
- spec-279 — the @mindset-ai/db-schema package (Backstage's
prerequisite)
- spec-280 — the Backstage shell
```

When work on one spec turns out to depend on code that lives on another spec's unmerged branch, the agent does not guess. It surfaces the dependency and offers to pin it as a decision that gates the build:

```
spec-389 generalizes spec-360's code, which is on an unmerged
branch (worktree-spec-343-scaffold-redesign), not develop. How
should we handle that dependency?

   1. Capture as a new decision (Recommended)
      Surface a new decision (D-5) pinning that spec-389's
      build is sequenced
      AFTER spec-360 merges to develop — making the hard
      dependency explicit on the Spec so it gates the build
      move.

   2. Note in narrative only
      Fold the dependency into the Overview prose as a
      sequencing note, without a formal decision. Lighter, but
      doesn't gate the build transition.
```

The same memory does something quieter, and often more valuable: it keeps the record honest when reality moves past it. When an agent's change contradicts a decision the team has already recorded, it does not silently override it. It flags the contradiction, fixes what it can, and tells you the record itself needs to catch up:

```
One thing to flag

This reverses spec-15 dec-3 / ac-9, which explicitly specified
the flip as "label-only, href never changes." I updated the
related test and dropped the now-contradicted ac-9 tag, but
the Memex spec itself still records the old decision. Per the
repo's Memex workflow, that decision/AC should be updated to
match.
```

That is drift detection from the inside. The agent could have made the change and moved on, leaving the spec quietly lying about what the code now does. Instead it refused to let the record drift out of sync with reality, and handed back the one judgement only you should make: whether the decision itself has changed.

Part Two — Getting Started

You now understand the system. Reading about a closed loop and feeling one close are different things, and the gap between them is one real spec. Memex runs through your coding agent, so this assumes you already work with an MCP-capable one such as Claude Code or Cursor. Here is the path, in five steps.

1. Create an account. Sign up at memex.ai. Everyone gets a personal Memex straight away. To bring your team in, create an Org Memex and add them. The free hosted plan includes three Org Memexes with unlimited users; host Memex yourself and you get unlimited Memexes too.

2. Connect your coding agent over MCP. Install the MCP connection to whichever coding engine you use, following the connection instructions. This matters more than it looks. The MCP connection is what grounds every spec in your actual code, so the work and the record never drift apart.

```
# Memex MCP or see the instruction link above
https://memex.ai/mcp
```

3. Teach your agent the workflow. Memex works best when your coding agent knows it is there. Point the agent at the MCP and let it update its own instructions file (`CLAUDE.md`, `AGENTS.md`, the Cursor rules, whatever yours uses). One prompt does it:

```
# Memex MCP or see the instruction link above
https://memex.ai/mcp
```

I want to use the Memex MCP. Examine its tools and update claude.md so it becomes part of my daily workflow.

From then on the agent reaches for Memex without being reminded.

4. Create your first spec. Pick a small, genuine piece of work, not a toy, and start a spec from inside your coding agent:

```
I want to create a new spec in Memex for...
```

5. Drive the loop. A spec moves through five phases, and once they are familiar, running Memex is simply driving them through your agent. Most of the work lives in the first; the rest is mostly the agent's.

- **Draft.** A rough idea, nothing committed yet. The holding pen before the real work starts.
- **Specify.** Where you do the real work: evolve the narrative, *resolve the decisions, resolve the comments.* Each decision you resolve auto-creates the acceptance criteria that enforce it, and a readiness check blocks the spec from moving forward until the decisions, criteria and comments are clean.
- **Build.** Set up the worktree and let the coding agent do the work across the tasks, with the full run of its sub-agents and tools. You settled the hard questions in Specify, so the code is its job, not yours. An umbrella spec can oversee several child specs when the work is large enough to need it.
- **Verify.** The checking phase: confirm the tests are sound and clear any issues Build surfaced. "Done" becomes a fact derived from your tests, not a judgement call.
- **Done.** Your call. Once you are satisfied, move the spec there and it is closed.

The Memex MCP drives the next actions from there, and you steer it in plain language:

```
I want to work on <paste the spec URL>

what decisions do we need to make on this spec

lets work through the decisions

lets resolve all open comments into decisions

slack this spec to Sally and ask for her architectural input

what have we decided in the past which is similar?

move to build and implement this spec in a worktree, using
sub-agents until it is complete

move this spec to done and send a slack message to the team
with an update
```

The hard questions

A system like this invites sharp questions. Here are the ones worth taking seriously, answered honestly.

Does decision extraction actually work, or does reviewing noisy candidates just recreate the manual logging you set out to kill?
This is the real dependency: if extraction surfaced junk, you would be back to hand-curating a log. Two things keep it honest. Reviewing a surfaced candidate is far cheaper than recalling and writing a decision from a blank page; you are accepting, rejecting or editing, not authoring. And a candidate you ignore costs nothing, because it does not block the work. The bar is not perfection, it is "better than the nothing most teams record today", and that bar is low. Extraction quality is a real variable and it improves as the models do, but the worst case is a missed decision, not a corrupted one.

Is six record types more schema than the work needs?
Possibly, for a small spec, and Memex does not make you populate all six. Narrative, decisions, criteria and tasks carry the real weight. Issues are thinner: they exist because an agent that gets stuck needs a structured place to put a problem that is neither a decision nor a task, so it surfaces to you instead of dying in a transcript. If that feels like

ceremony on a trivial spec, use fewer records. The schema is a ceiling on structure, not a floor.

Passing tests is not the same as being right. Does "done derived from tests" overclaim?
It would, if we said tests prove correctness. They do not. Tests define necessary conditions, not sufficient ones, and for anything with feel, emergent behaviour or underspecified edges they fall short. "Done derived from tests" means one specific thing: the conditions you agreed mattered are still met. The human still judges whether the result is actually right; Memex only removes the lie that something was checked when it was not. The system is strongest for well-specified work and stays deliberately in your hands where specification is hardest.

If the loop only checks the criteria a human wrote, can it close confidently around the wrong thing?
Yes. A derived "done" is only as honest as the criteria feeding it, and under-specified criteria will pass a build that is wrong. Memex does not automate that judgement away; it moves it, from "did anyone verify this?" to "did we ask for the right things?". That is a better place for the question to live, because it sits visible and reviewable on the spec rather than buried in one person's confidence. Writing good criteria is still your job. We think that is the right division of labour, not a problem we have solved.

Is front-loading the Specify phase just waterfall in agent clothing?
It would be, if the spec were frozen before building and never revisited. It is not. The spec is fed by the build: an agent meets an unforeseen choice and writes a candidate, hits a wall and files an issue, and drift detection flags the decisions that the build outgrew. You specify enough to start, not everything upfront, and the loop carries what you learn back into the record. Drift is not an embarrassment to the model. It is the part that makes building in order to learn safe.

We hope you enjoy using Memex and get involved in the community. Remember Memex is open source and we welcome PRs.

Come and say hello on Discord, whether you are weighing Memex up or already building with it.