



POSSIBILITY SPACE • FULL USE CASE

02

An agent that remembers, and can prove it

A memory the agent keeps outside any app, where every entry is signed and every past entry stays readable.

BUILD STATE

Yours to take

COMPLEXITY



Vision

An assistant that has been given the same correction twenty times still opens the next session knowing nothing about it. What makes an agent useful accumulates slowly: the decisions already settled, the approaches already tried and dropped, the particular way one person wants their work done. That accumulation is the part nobody wants to retype every morning, and it is the part that isn't kept.

Where memory does exist, it is a feature of one product. It sits in a vendor's database, reachable through that vendor's interface, on that vendor's retention schedule. Moving to a different model means starting from nothing, because the history was never something the person it describes actually held.

Memory stops being a convenience question the moment agents start acting. An agent that files an expense, sends a message, commits code or moves money leaves a record behind it, and when someone asks afterwards what the agent knew at the time, that record is the answer. Right now the party that operated the agent is also the party that writes and stores that record, and nobody outside can check it. For an agent working inside a business, or on behalf of a client, that gap gets expensive.

Swarm changes where the memory sits and who is able to check it. Entries go into storage that no company operates, and each one is signed by the agent's own key. The address of any entry follows from three public facts: the agent's address, the name of the memory, and which entry it is. Anyone can work out that address and read what is there. Only the holder of the key can add to the memory, nobody else can forge an entry inside it, and the agent cannot go back and rewrite its own past either.

The memory also outlives the app that created it. Point a different model, a different framework or a different machine at the same address, and the history is still there, because it was never inside the app in the first place. Storage is paid for upfront instead of billed monthly, so there is no account whose closure ends it. None of this is far from where the tooling already is: Swarm's MCP server exposes the feed operations an agent needs as ordinary tool calls, so a logbook an agent writes to and updates is a handful of calls rather than a research project.

Architecture

A memory on Swarm is a feed: a named, append-only log that one key writes and anyone can read. Every mechanism below serves those two properties.

Content addressing (shipped). Data uploaded to Swarm is split into chunks, each addressed by the hash of its own contents, and spread across the network's independent nodes. An address is therefore a statement about content rather than a location: fetching it tells you whether you got exactly what the address names.

Feeds and Single Owner Chunks (shipped). A feed is a series of Single Owner Chunks signed by an Ethereum keypair. Two values identify one: the owner's address and a topic, where the topic is a 32-byte value derived from whatever name the memory is given. The name is the agent's own choice, nothing is reserved or predefined, and the same name always derives the same topic, so a memory called something like `agent_log` or `project-notes` has an address that can be worked out again later instead of a random number somebody has to keep track of. Each update is written at the next index in sequence. A chunk's address is the hash of its topic-and-index identifier together with the owner's address, which means the location of entry 41 of a given memory follows from public information alone. No index service, no registry, no gateway that has to cooperate.

What “verifiable” rests on (shipped). Each entry is a chunk signed with the agent’s private key, so a reader can confirm both that it came from that key and that it arrived unaltered. Nobody else can append to the memory or fabricate an entry inside it.

Entry size (shipped). An entry up to a chunk’s worth of data lives in the feed chunk itself. Anything larger is stored as ordinary content-addressed data with the feed chunk wrapping its root, so a long entry and a short one are written the same way and no size ceiling has to be designed around. A bulky artifact that the memory only refers to, a transcript, a dataset, a document, can also be uploaded on its own and referenced by address, which keeps the memory itself small and quick to read.

Postage stamps (shipped). Storage is prepaid. A batch is bought with a capacity and a duration and it funds the nodes holding those chunks for as long as it lasts; topping it up extends the memory’s life. This is what stands in for a subscription.

Private memory with ACT (shipped in Bee). A feed written in the open is readable by anyone who knows the owner and the topic. Swarm’s Access Control Trie encrypts content at chunk level and maintains a grantee list of public keys, with each grantee’s copy of the access key encrypted specifically for them, derived from their own public key. The publisher adds and revokes grantees. Revocation applies going forward: a removed grantee keeps access to the versions it could already read, which suits an audit trail, and rules out using revocation to change what somebody already saw.

Reaching all of this from an agent (Model Context Protocol, external). The Model Context Protocol is an open standard for letting a model call tools, and it is not Swarm’s; it comes from outside the project. What Swarm has is a server that speaks it. The Swarm MCP server implements Swarm’s operations as tools a model can call: storing and fetching data, writing and reading a feed, buying and extending the prepaid storage that keeps it alive. An agent therefore reaches a memory the way it reaches any other tool, deciding for itself when to write an entry and when to read one, with no Swarm-specific code of its own. ACT is the one part not yet in a release: the tools for encrypting an entry and managing its grantee list are written and due with the version of the server that adopts the 2026-07-28 revision of the protocol. Until then a private memory is reachable through Bee directly rather than through MCP.

What Swarm does not do. Storing entries and handing them back is the whole of it. Deciding what is worth remembering, compacting a history that has grown long, and searching it by meaning are the agent’s own work, with whatever embedding model or index its builder prefers. Nothing in Swarm ranks, summarises or interprets anything.

PRIMITIVES

Content addressing

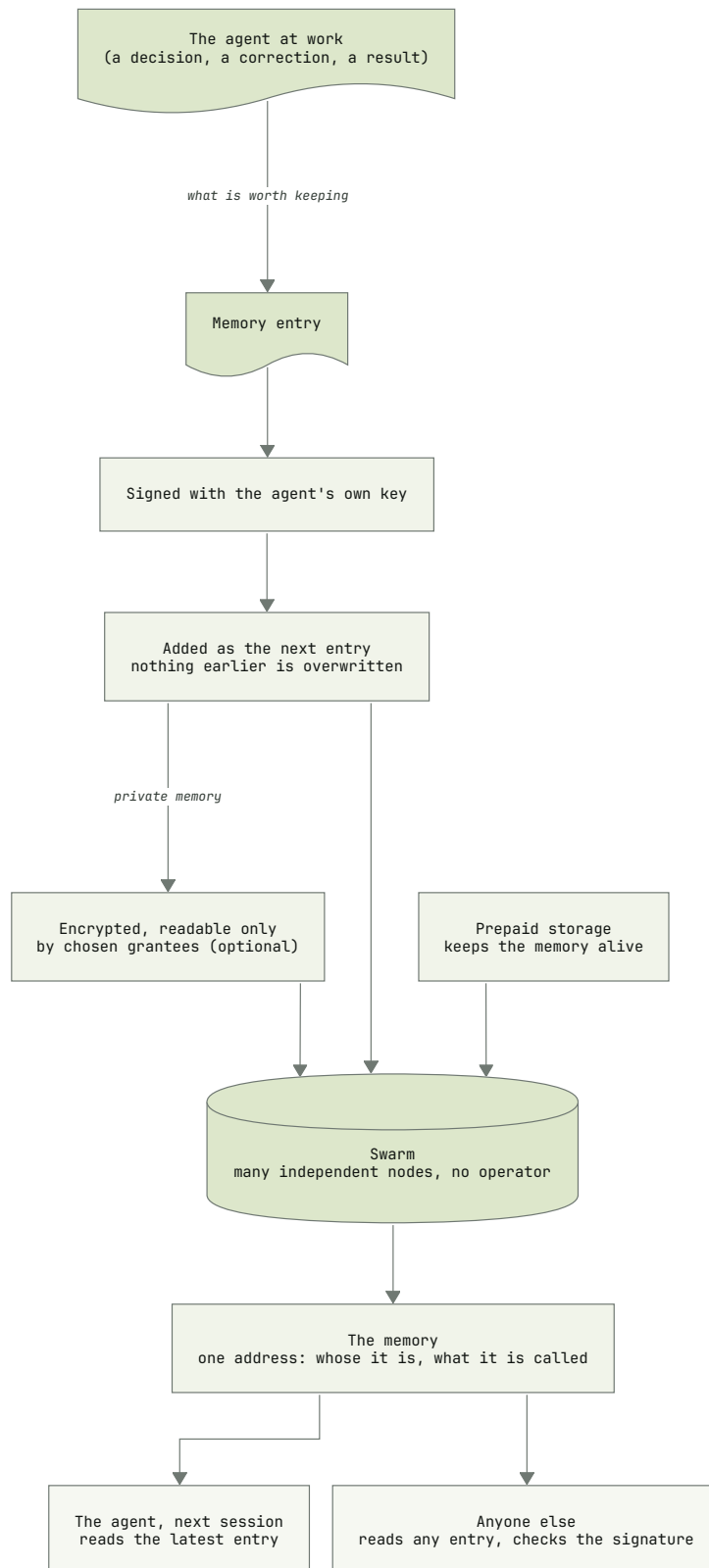
Feeds / Single Owner Chunks

Postage stamps

ACT

EXTERNAL DEPENDENCIES

Model Context Protocol



Data flow. The agent picks out what is worth keeping; the entry is signed with its own key and added as the next entry, leaving earlier ones untouched; it is stored across Swarm's independent nodes and kept alive by prepaid storage. The memory has a single address, and from there the agent reads the latest entry while anyone else can read any entry and check the signature. An optional branch encrypts entries so that only chosen grantees can read them.

Development

The short path is the Swarm MCP server, which presents Swarm as ordinary tool calls: four for paying for storage, five for moving data, two for the memory feed, one for tracking a slow upload. Setting it up is most of the integration; after that the agent has Swarm as a capability and decides for itself which tools to call and when, the same as with any other MCP server. The memory pattern below is what falls out of those tools rather than something that has to be wired in advance, and an agent that can call MCP tools can use any of it without knowing what a chunk is.

Running it. `npx swarm-mcp` starts the server, or clone the repository and build it. Two environment variables carry the weight. `BEE_FEED_PK` is the private key that is the agent's identity, and the only key that can write its memory. `BEE_API_URL` points at a Bee node and falls back to the public gateway, which is enough to get a first memory written before running a node. `AUTO_ASSIGN_STAMP` is on by default and picks a funded postage batch for uploads so the agent never has to choose one. For a desktop MCP client the server runs over stdio; for a server-side agent or a container it serves HTTP at `:3000/mcp`, with the filesystem tools available in stdio mode only.

The memory loop. Writing is `update_feed` with data and a `memoryTopic`; reading is `read_feed` with the same topic. The topic is any plain string, hashed to 32 bytes by the server, so names the agent picks itself work as topics, `agent_log` or `project-atlas/decisions` for instance, and one agent can keep several memories apart by naming them differently. Each `update_feed` call appends at the next index. For anything bulky, `upload_data` returns an address and the entry carries that address instead of the payload.

Keeping it alive. `create_postage_stamp` takes a size and a duration, `list_postage_stamps` shows what is currently funded, and `extend_postage_stamp` adds capacity or time to a batch already in use. A long-running agent needs this on a schedule, and it is the one operational job with no equivalent in a hosted memory feature: the memory expires when its batch does. Buying the batch as immutable is what makes the memory unrewritable afterwards, including by the agent itself.

Letting someone else check. `read_feed` takes an optional owner, so anyone else holding the agent's address and the topic name can read the memory without the agent's private key or cooperation. That call returns the latest entry only. Reading a specific earlier index means going through bee-js or the Bee API directly, where any past entry's address is computable from owner, topic and index; the MCP tool has no index parameter. ENS names are not accepted in place of an owner address yet, so pass the raw address.

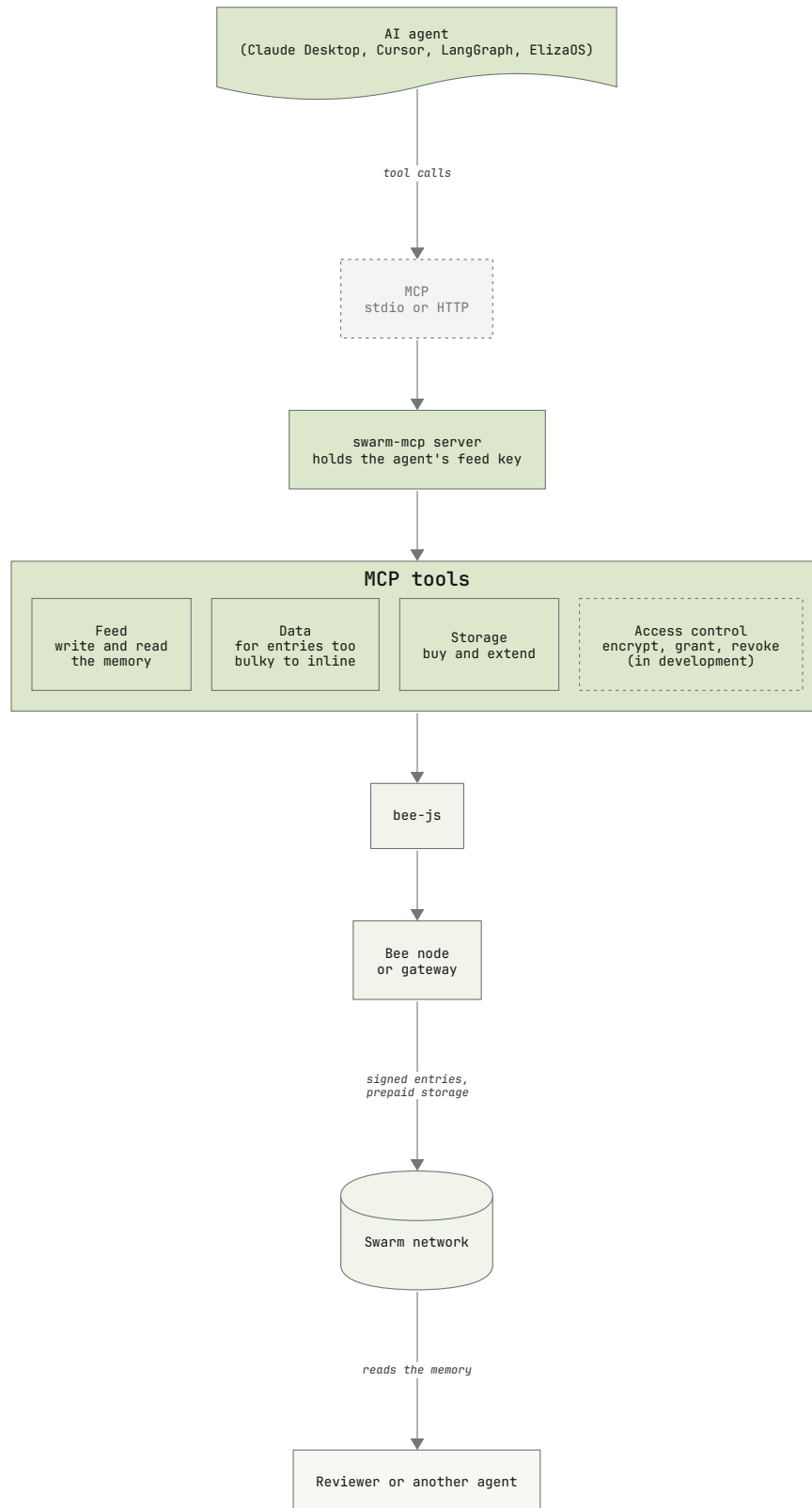
Private memory (in development). ACT is the one piece the released MCP server does not cover yet, and it is close rather than absent. The tools are written and waiting on a release: creating a grantee list, listing it, adding and revoking grantees, and reading back the Bee node's ACT publisher key, plus `act`, `grantees` and `historyAddress` parameters on the upload tools and `actPublisher`, `actHistoryAddress` and `actTimestamp` on `download_data`. That release is the one adopting the 2026-07-28 MCP specification, so plan for it rather than build on it today. Worth knowing what it covers: the feed tools take no ACT parameters, so a private memory is composed rather than configured. Encrypt the entry body with an ACT upload call and write the returned reference into the feed, which leaves the feed itself readable while its contents are not. The node's ACT publisher key is what one agent hands another before it can be added as a grantee. Until the release lands, the same work goes through `swarm-cli grantee create / patch / get` and Bee's `--act` upload flags, or bee-js

in the agent's own code. One documented quirk to design around either way: a grantee-list update called within a second of the previous one fails with an invalid input error.

Other framework paths. For an ElizaOS agent, swarm-elizaos-agent wires the same operations in without MCP in the middle. For anything custom, bee-js is what both of those are built on.

TOOLS & REFERENCES

- swarm-mcp — <https://github.com/Solar-Punk-Ltd/swarm-mcp> (published as swarm-mcp on npm)
- swarm-elizaos-agent — <https://github.com/Solar-Punk-Ltd/swarm-elizaos-agent>
- Bee Feeds documentation — <https://docs.ethswarm.org/docs/develop/tools-and-features/feeds>
- ACT developer guide — <https://docs.ethswarm.org/docs/develop/act>
- Access control concepts — <https://docs.ethswarm.org/docs/concepts/access-control>
- Postage stamps — <https://docs.ethswarm.org/docs/concepts/incentives/postage-stamps>
- bee-js — <https://bee-js.ethswarm.org>
- swarm-cli — <https://github.com/ethersphere/swarm-cli>
- swarm-ai-data-exchange — <https://github.com/Solar-Punk-Ltd/swarm-ai-data-exchange> (prototype; agent identity and payments on the same feed and ACT machinery)
- **External dependency (not a Swarm primitive):** Model Context Protocol (modelcontextprotocol.io); the ACT work targets the 2026-07-28 revision of the specification



Component assembly. The agent reaches the swarm-mcp server over MCP, on stdio or HTTP; the server holds the agent's feed key and drives bee-js against a Bee node or a gateway. Four groups of tools matter: the feed tools that write and read the memory, the data tools for entries too bulky to inline, the storage tools that buy and extend the prepaid batch, and the access control tools, which are still in development. A reviewer or another agent reads the memory straight from the network.

Reference implementations

Yours to take. The tooling for this is shipped and published; no production system has been built on it yet that could be verified, which is why this space is open rather than claimed.

swarm-mcp (Solar Punk) — <https://github.com/Solar-Punk-Ltd/swarm-mcp> — the MCP server described throughout this document, published to npm as `swarm-mcp` (0.9.5, August 2026). Twelve tools covering data, feeds and postage stamps, over stdio or HTTP, with a Dockerfile. Real, working, maintained infrastructure rather than a demonstration. ACT support, covering grantee management and ACT parameters on the upload and download tools, is written and due with the release that adopts the 2026-07-28 MCP specification.

swarm-elizaos-agent (Solar Punk) — <https://github.com/Solar-Punk-Ltd/swarm-elizaos-agent> — the same uploads, feed reads and updates, and stamp management, wired into ElizaOS for agents built on that framework.

swarm-ai-data-exchange (Solar Punk, hackweek v1, April 2026) — <https://github.com/Solar-Punk-Ltd/swarm-ai-data-exchange> — a prototype marketplace where agents publish and buy data, combining Swarm and ACT with ERC-8004 identity and x402 payments, both external to Swarm. Labelled a prototype by its own authors, and its ACT and marketplace-feed MCP tooling lives in a pinned draft fork of `swarm-mcp`; ACT has since been implemented properly in `swarm-mcp` itself. Relevant here as the same feed-and-ACT machinery aimed at a different problem.