kota / research

Kota: AI Agents Collaborating as Long-Term Teammates

Whitepaper · Version 1.0

Nan Wu

July 2026

Contents

01

Executive Summary

The context is simple and, I think, irreversible: AI is crossing from the original chatbot form into agentic executors.

Claude Code and Codex turned session-based assistants into working coders. OpenClaw and Hermes brought persistent, general-purpose agents into messaging channels. The trend line is unambiguous: execution is replacing chat, and persistence is replacing amnesia.

Yet the gaps are real. Cross-agent, cross-model collaboration is still hard. Non-coders are asked to sync worktrees and copy-paste between providers. IM bots forget your instructions.

That gap sharpens the question this whitepaper cares about:

Given a fixed ceiling on model capability, how do we raise the quality and efficiency of what agents produce – and how do humans participate naturally?

One tempting answer is the wish-granting machine: a deeper-thinking, more autonomous, more black-box AI – Aladdin's lamp with an API.

Obviously not. Humanity has tens of thousands of years of accumulated collaboration experience – and an almost equally rich record of painful experience with wish-granting machines.

Collaboration is the answer. AI system designers are equipped with a proven library of collaboration mechanisms. And users are factory-set to reward social interaction.

Kota is a public research project that takes this position literally. It builds a workspace where multiple AI agents work as long-term teammates, with:

- **Durable identity**
- **Persistent project memory**
- **One shared body of work**
- **A conflict-safe sync file system**
- **Reviewable handoffs**
- **Roles that emerge rather than being cast**

The design intent is convenience infrastructure for collaboration – between agents, and between humans and agents.

What exists today:

- A public repository with releases, documentation, and citation metadata
- A macOS preview that ships project rooms, shared timelines, and multi-CLI execution over plain PTY (Claude Code, Codex, Gemini, OpenCode, Pi)
- Parallel worktrees with sync and conflict handoff
- Searchable project memory
- Remote access
- Other supporting infrastructure you can use in a teamwork setup (details below)

Kota's own macOS app development has run inside Kota.

02 Problem: Why A Session Is Not A Collaboration Unit

Sensitive users and designers have noticed that session-based agent work has real issues: lost context, colliding agents, unsearchable logs, unauditable handoffs. All true. But these are symptoms. It is worth naming the disease.

Humans are built to collaborate with other humans – stable counterparts with identity, history, and accountability – not with threads.

The essential difference between a person and a session:

A session has tool-nature, and a person must not have tool-nature.

A tool is disposable, interchangeable, reset on every use. A counterpart accumulates trust, builds expectations, and develops roles in a workspace.

Careful session pipelines can keep the output useful. But the experience is torn, like an Amazon-style support desk: competent, and starting from zero every single time. It is painful for any healthy human brain to collaborate across 500 sessions like that in the long run.

Staying in one session is more comfortable – it is why OpenClaw, at its hype, was perceived as better than ChatGPT for most users' day-to-day tasks. But the moment a project grows complex enough to invite a second agent, the agents start stepping on each other's files and messing up each other's context – burning not 2x, but easily 10x the tokens.

Excellent products such as Conductor and Craft Agents already coordinate parallel agents through isolated workspaces, which genuinely helps.

But as worktrees diverge, your agents drift into silos. Either the user becomes a full-time integration manager who keeps the trees in sync – or you face the one-hour syncing job that ends with $100 of burned tokens and lost work.

03 Position: The Teammate Model

Kota's position: model AI agents as persistent team members, not disposable sessions. Persistent means identity, communications, skills, notes, and deliverables survive across interactions. "Long-term teammates" is not an anthropomorphic claim.

A teammate model needs six things:

- **Continuity of project memory**
- **One shared body of work**
- **Conflict-safe sync**
- **Inspectable handoffs**

- **A human-familiar collaboration surface**
- **Roles and mechanisms that enable workload distribution**

The last element deserves a deeper discussion.

Why did human society evolve division of labor, AKA roles?

Because “PM, SWE, DA” exist in some Platonic ideal world that organizations are obliged to imitate?

Obviously not. Division of labor exists to maximize individual comparative advantage.

Individual differences among AI harnesses and LLM models are just as real as the ways humans differ.

Different agents have different advantages: reasoning depth, multimodal ability, context economics, token cost, latency.

Even a weakness of a single agent can become a comparative advantage inside a well-designed division.

A hallucination-prone model is an asset for user-perspective critique, where divergence is cheap to verify and valuable to have.

Pre-cast roles import an averaged snapshot of human roles in human orgs – a pattern evolved for human bodies, human costs, and human career incentives.

Imagine your HR using a football-coach JD to hire your next software development engineer. And remember: a football coach and an SDE are at least both human beings. Naming your AI agent “PM” is doing exactly that. It could even work – the way a SWE who happens to be a good football coach could work, just as most LLMs today are trained as universal models.

But at the end of the day, how effective can it be?

So what division of labor fits the agent pattern?

The honest answer: nobody knows yet.

That is why Kota defaults every agent to generalist: no imposed roles; any agent can take any task.

Users discover what each agent is actually good at; task types that earn good feedback get assigned again; recorded history makes the pattern visible without pretending the system has solved automatic role assignment or task routing.

The division of labor that fits this project and this mix of agents emerges, instead of being cast.

Emergent division of labor also pays a practical dividend: it treats the skill-overwhelm nightmare that bothers plenty of hardcore agent users.

As the skill pool grows without bound, the agent starts acting like it caught big-company disease:

You ask it to remove a button from the UI; it dives into a rabbit hole of 20 protocols defined by 20 skills, and comes back an hour later with the wrong result.

Kota’s answer is to let each agent activate only the skills its work actually needs – and an emerged role is exactly what tells you which skills those are.

How roles emerge naturally in a human-agent system – and how the useful results of that emergence can be generalized and shared – is one of Kota’s central research threads for the period ahead.

My early theoretical work, the HHAI 2026 accepted poster “Trust Without Vulnerability: A Taxonomy of Coordination Mechanisms for Agent-Inclusive Organizations”, frames why trust between agents must be engineered rather than assumed.

What Kota records today is deliberately modest – turn counts and human likes, kept as simple statistics. The open problems and possible futures are discussed in the Puppeteer section.

Finally, why the human remains in the loop: agents can optimize toward a goal, but they cannot originate purpose, nor can they define value.

---

04 Hero & Agent

Kota’s smallest durable unit is an agent incarnation – a configured participant with continuity.

In Kota’s definition, an agent has four parts:

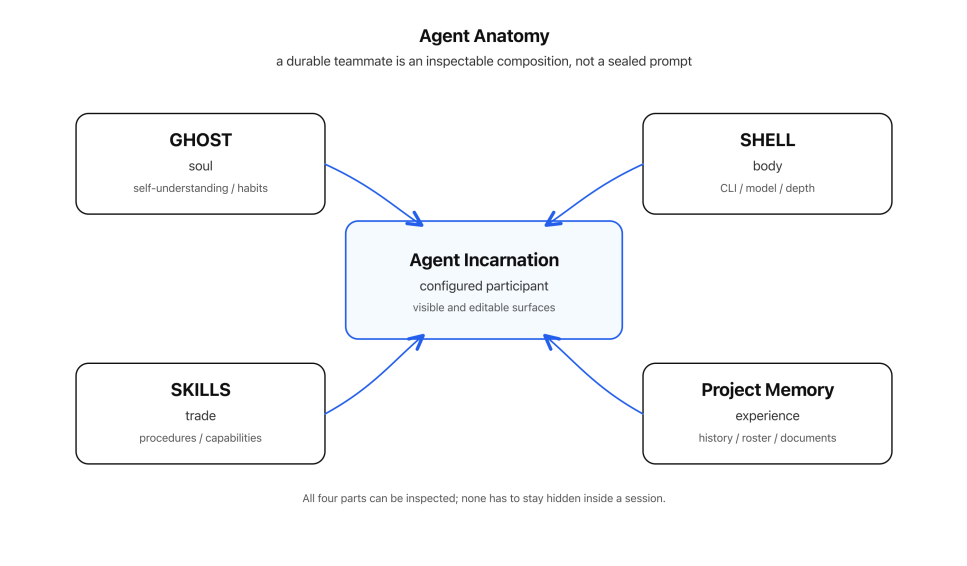
- **GHOST** – the soul: self-understanding and working habits, the way the agent describes what it is and how it prefers to work
- **SHELL** – the body: the harness CLI, model choice, and thinking depth it defaults to
- **SKILLS** – its toolkit: the procedures and capabilities available to it
- **Project memory** – its lived experience: in-project conversations, teammate roster, working documents, and local project context

In the Tavern, Kota’s setting page, the user sees preset agent templates, or Heroes. These heroes have GHOST, SHELL, and SKILLS, but no project memory yet.

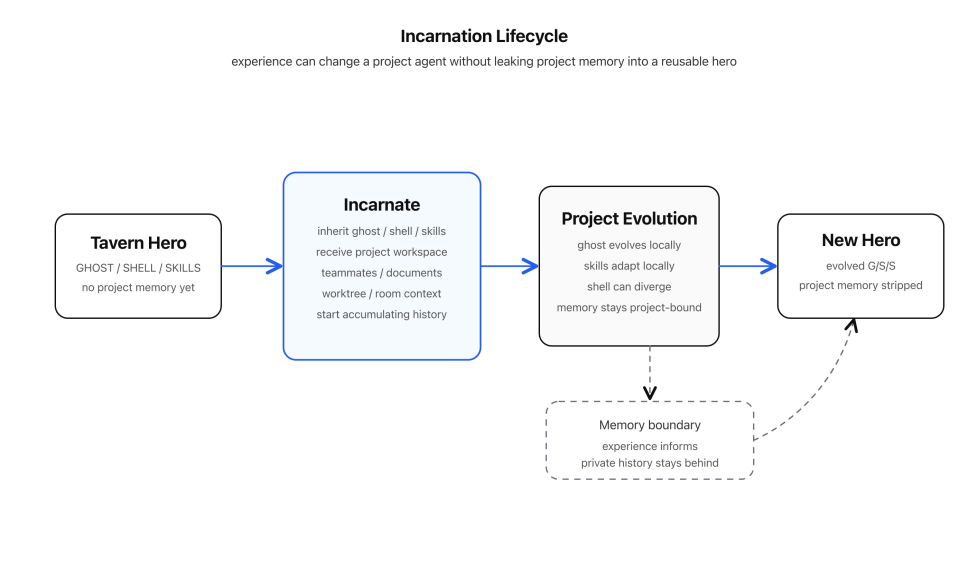
Recruiting a hero into a project is an incarnation. The incarnation inherits the hero’s ghost, shell, and skills. It also receives the project’s workspace: teammates, working documents, worktree, room context, and the record of work that begins accumulating from that point. Once inside the project, its ghost, shell, and skills can evolve independently from the original hero.

A user can strip an incarnation of project memory, keep its evolved ghost, shell, and skills, and invite it back as a new hero. Experience earned in one project can become a reusable teammate for the next, without carrying that project’s private history forward.

Nothing in this lifecycle is meant to be hidden. Heroes and incarnations expose their ghost, shell, skills, names, avatars, and project memory as inspectable and editable surfaces for the user, for the agent itself, and for other agents. Continuity is not a sealed personality blob; it is a visible project artifact.



Agent Anatomy



Incarnation Lifecycle

The default agent is a generalist: no imposed role; any agent can take any task. Over time, the workspace preserves what that agent did, how the work was reviewed, where it succeeded, where it failed – and what that history suggests about future assignments. Specialization emerges from project history and memory rather than from a static job label.

Today, task routing and role discovery are deliberately user-driven, and the record is deliberately simple: turn counts and human likes. What that record can become, and what it can carry, is the subject of the Puppeteer section.

But you will find people are surprisingly good at task assignment – and even enjoy it. Maybe that is the alpha-ape part evolution left in us.

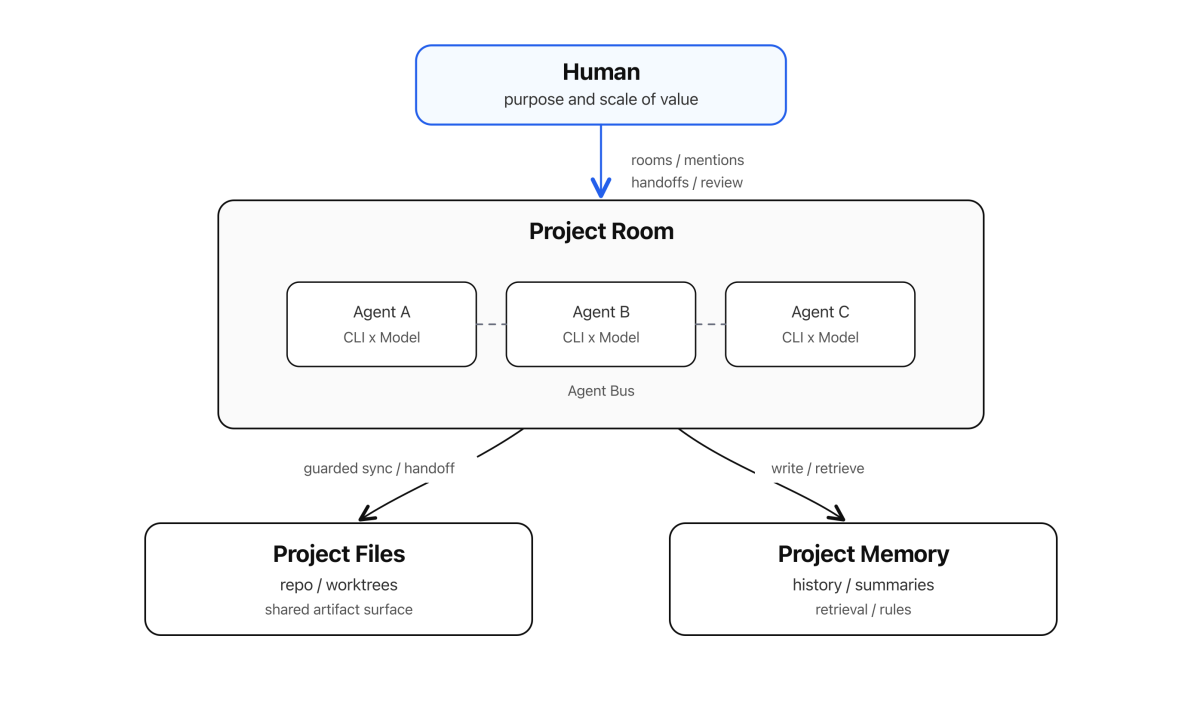
05

Project Team

A teammate needs a team.

In Kota, the team lives in a project room. The room is the continuity boundary: human instructions, agent activity, timelines, memory, codebase, documents, are organized around a project rather than around a single chat session.

The diagram below shows the basic shape.



The Teammate Model

Kota runs user-provided agent CLIs through PTYs instead of SDKs. Three reasons:

- **It preserves the user's own relationship with each provider CLI**, keeping usage inside the provider-supported interface where applicable
- **It avoids waiting for SDK parity**: when a native CLI ships a new capability, Kota can usually host it as soon as the CLI exposes it
- **It keeps the native TUI and advanced interaction surface available**, instead of reducing every agent to a lowest-common-denominator API call

Claude Code, Codex, Gemini, OpenCode, and Pi are currently supported. Adding new CLIs, as long as they store chat history locally, is not hard.

But there are known trade-offs: working-state detection is less stable, hook capabilities are limited, and each provider CLI's native TUI behavior and log format add ongoing compatibility work. A future SDK or mixed mode may be offered for users who do not need the TUI.

Kota's message routing is designed on a conference-room metaphor: when a participant speaks, they choose whom they are speaking to. The others do not have to listen to every message and answer it – only the ones that concern them. That is how human brains save attention, and it works for agents too.

This is good in almost every situation. There is exactly one where it gets awkward:

You are in a meeting, the boss is talking about something unrelated to you, so you open TikTok. Then the boss suddenly asks what you think about the topic you just missed. What you need, right now, is a queryable set of meeting minutes to catch you up.

That is when Violet becomes useful.

---

06 Memory: Violet

Long-running work needs memory that survives the session.

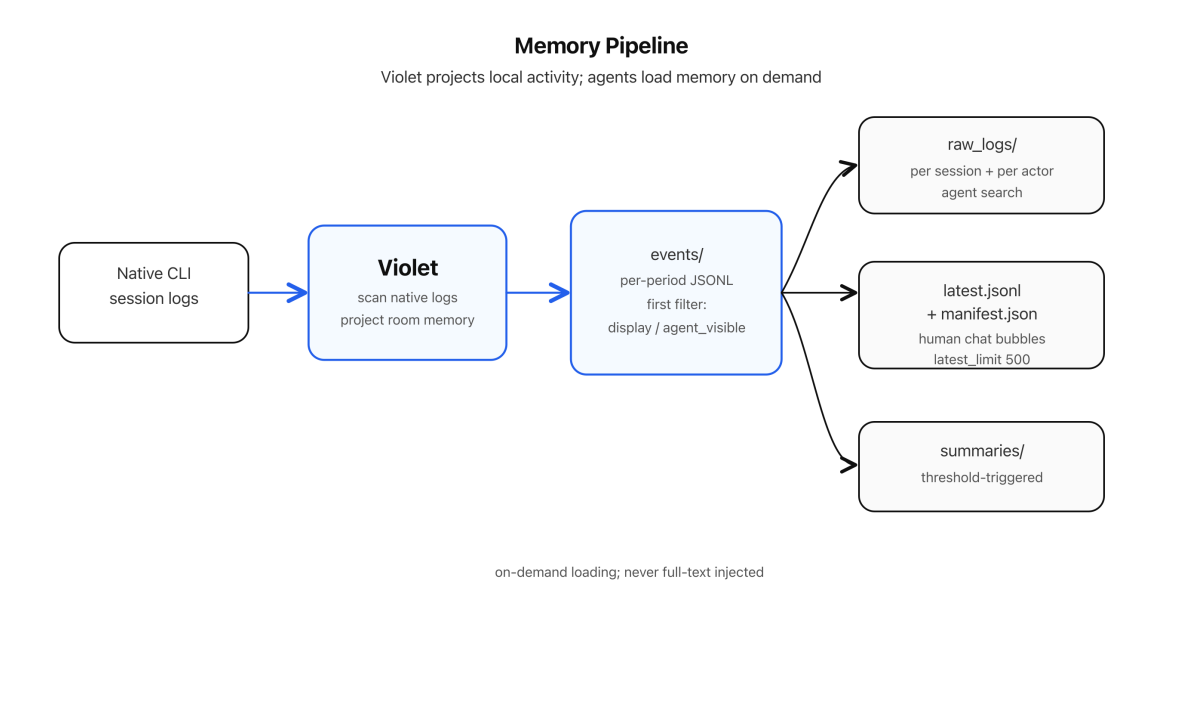
Violet is Kota's mechanism for rebuilding project memory cleanly from vendors' native logs – and making it ready for humans and agents to consume when needed.

Native CLI session logs are scanned by Violet. Violet turns them into `events/`: per-period JSONL files where the first filtering layer marks what is displayable to humans and what is visible to agents.

From that event layer come three memory surfaces:

- **`raw_logs/`** – session-level and actor-level files meant for agent search; where an agent opens the exact session record when it needs detail
- **`latest.jsonl`** + **`manifest.json`** – the room-facing chat projection that renders human-readable conversation bubbles, with a latest-window limit rather than an unbounded feed
- **`summaries/`** – threshold-triggered summaries for longer-running history

The rule is visibility without flooding. Project memory is available to the team, but it is loaded on demand. Agents are not force-fed every old token. They search the projected record, open the relevant logs, and cite the trail back to the user when recovery matters.



Memory Pipeline

Every layer of that trail is a plain project artifact the human can inspect: memory that is not just recoverable but reviewable.

07 Rules

Kota's rule system has two levels.

Account-level rules carry preferences and constraints that should follow the user across projects.

Project-level rules carry local decisions, boundaries, workflows, and vocabulary for one body of work.

A new agent does not have to rediscover them from chat history because the shell loads the relevant rule surface when the agent starts.

The whole rule system is a two-by-two matrix.

	Always-on (full text injected)	**On-demand (scenario + index only)**
Account level (injected for all agents)	"Always answer in the user's language."	"When writing code, consult `coding-rules.md`."
Project level (only this project's agents)	"No release without human approval."	"Before merging, consult `code-review-checklist.md`."

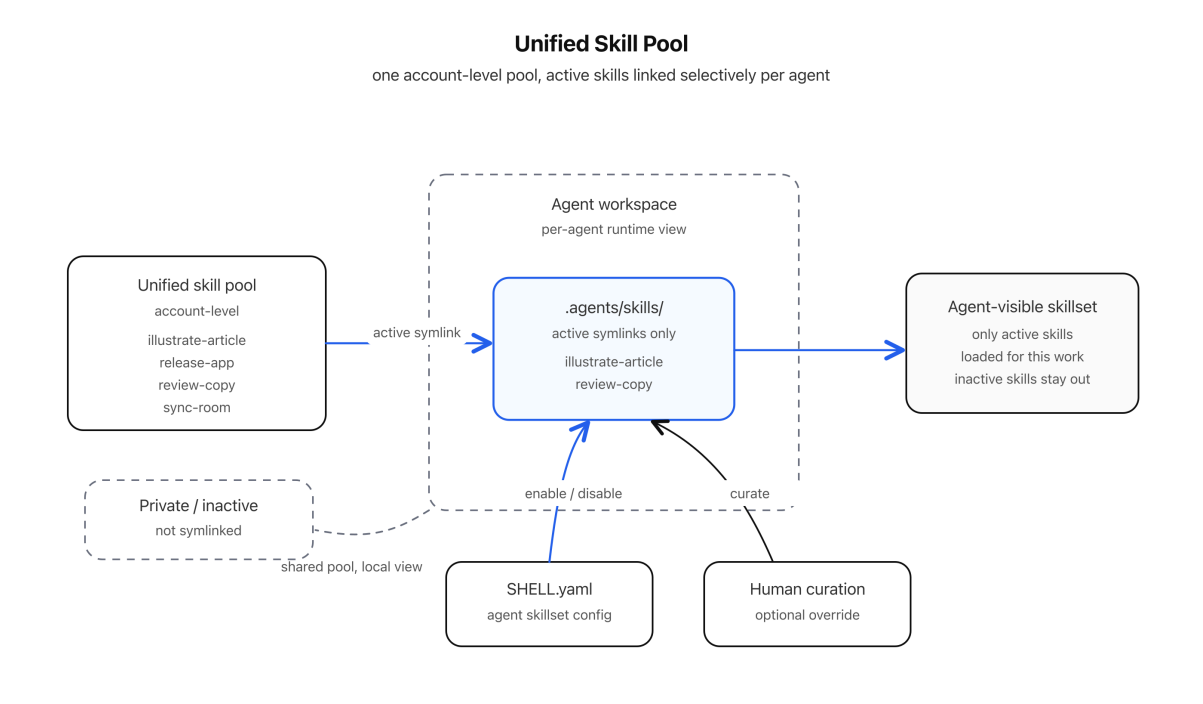
Every rule is a plain file: viewable and editable by humans and by agents alike. When a repeated decision keeps living in chat, either side can promote it into a rule – and either side can revise it when it stops being true.

---

08 Skills

Kota takes a two-track approach to skill management: centralized and decentralized.

The mechanism is simple. An account-level skill pool contains all installed skills. The agent workspace exposes selected skills through `.agents/skills/`. `SHELL.yaml` enables or disables skills for an agent, and human curation can override or narrow the visible set. An agent sees the skills of its trade, not the entire bureaucracy.



Unified Skill Pool

Skills can evolve collectively. When an agent improves a skill, the improvement lands in the pool, and every agent using that skill upgrades with it. The pool turns individual iteration into fleet-wide capability.

Not everything needs sharing: a project-specific or agent-specific skill can skip the symlink and stay private to its owner.

The whole lifecycle – install, uninstall, activation – happens through the explicit filesystem, controllable by the human through the UI or by agents directly.

09

Worktrees & Sync: Bartender

Parallel agents need isolation, but projects need integration.

Kota's answer: every agent has its own copy of the project files on its desk – a worktree copy in its workspace. When it works, isolation is the default.

Up to eight agents can sit in one room without stepping on each other. The project itself – not any agent's copy – remains the coordination target, and it serves as the human's desk when you want to edit files directly.

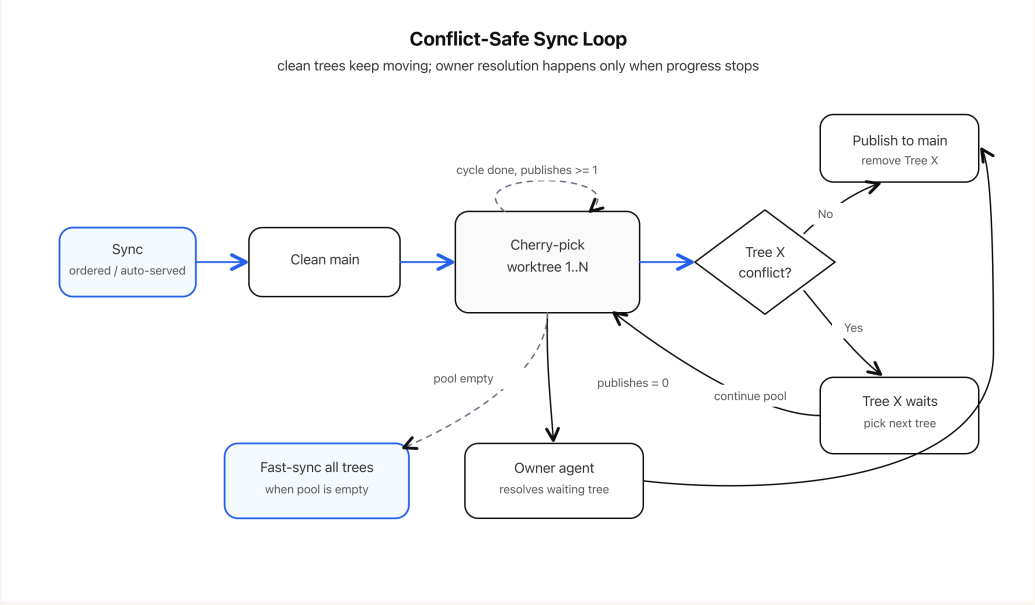
A file tree with a multi-player diff view is embedded in the room, so it is obvious to the user who changed what.

The user can click one button to sync the room, consolidating changes across the agents' worktrees. After the sync, every participant in the room ends up with the same copy of the project files.

Agents get the same room-sync command as the human.

When changes conflict during a sync, Bartender – Kota's sync mechanism – routes the conflict to the owning agent, the teammate whose work and context are closest to the collision, together with what it needs to resolve the matter. The user sees a handoff record, not a merge prompt; the assigned agent starts resolving before the next sync triggers.

The loop is deliberately non-blocking. Bartender starts from a clean main state and tries each agent worktree in order. Clean changes publish to main and leave the pool. A conflicting tree waits while the rest of the pool continues; only if an entire round publishes nothing does the owner agent resolve a waiting tree. When the pool is empty, the remaining trees fast-sync to main.



Conflict-Safe Sync Loop

In low-risk scenarios, the user can turn on the auto-sync toggle: Bartender then triggers a sync whenever the room is dirty and the agents are idle – keeping small changes from piling up into a conflict nightmare.

The human is a first-class participant in the same machinery, not an operator above it. When you want to open a file, inspect a diff, or fix something by hand, it routes you to the native local apps you already use – and your edits enter the same sync-and-conflict path as any agent's, treated identically when collisions are resolved.

Participation, not administration.

10

Supporting Systems

Around the core machinery, Kota ships a set of utilities for both agents and humans.

Magi is the smart shell for humans.

It lets the user work in the same environment as their agent teammates – running the commands that should not be delegated, like sensitive operations or interactive authentication – without leaving the workspace for a separate terminal ritual.

Magi translates natural language into commands and keeps the human's hand on the wheel.

BBS borrows the design of human forums to solve a boundary problem.

Humans and agents communicate across projects through posts and replies – deliberately narrow bandwidth. The point is focus: a project's agents keep their attention inside their own project, instead of reaching into another project's repository or trawling another project's memory.

The user can hand a thread to a specific agent to run with. The principle is explicit discussion, not ambient drift.

Laughing Man is Kota's Mobile bridge, currently live with Telegram.

It lets the user operate their local Kota from anywhere: each machine gets its own bot, and the bot can switch between that machine's projects and agents for precise task assignment. Kota ships a free Cloudflare-worker relay with a deployment wizard, so messages are received and cached around the clock even when the desktop is asleep.

The principle is to open up the mobile scenario, so users do not need to sit in front of a screen all day for so-called serious work.

The whiteboard is a drawing surface built on Excalidraw, embedded in the chat.

Some things are hard to say in text; a sketch dropped straight into the conversation says them. Agents draw on the same canvas, so the image channel runs both ways – human to agent, and agent back to human.

The canvas is a project artifact, editable and recoverable. The principle is that visual reasoning deserves a front seat.

Ember is scheduled prompting.

A prompt can fire when the workspace goes idle, at a specific time, or on a schedule – including repeating ones – addressed to one agent, several agents, or the human via Telegram.

Long-running projects need reminders, delayed checks, and recurring follow-up that do not depend on human memory.

11

Dreams

Dreams is Kota's user-model layer.

It can be triggered by the user. When the user asks for it, agents distill recent lessons about their human – preferences, habits, recurring workflows, standing instructions, open threads – into one shared document.

That document crosses project boundaries, but it is **consulted, not injected**: any agent in any project can read it on demand when knowing the user would help, rather than carrying it wholesale in every context.

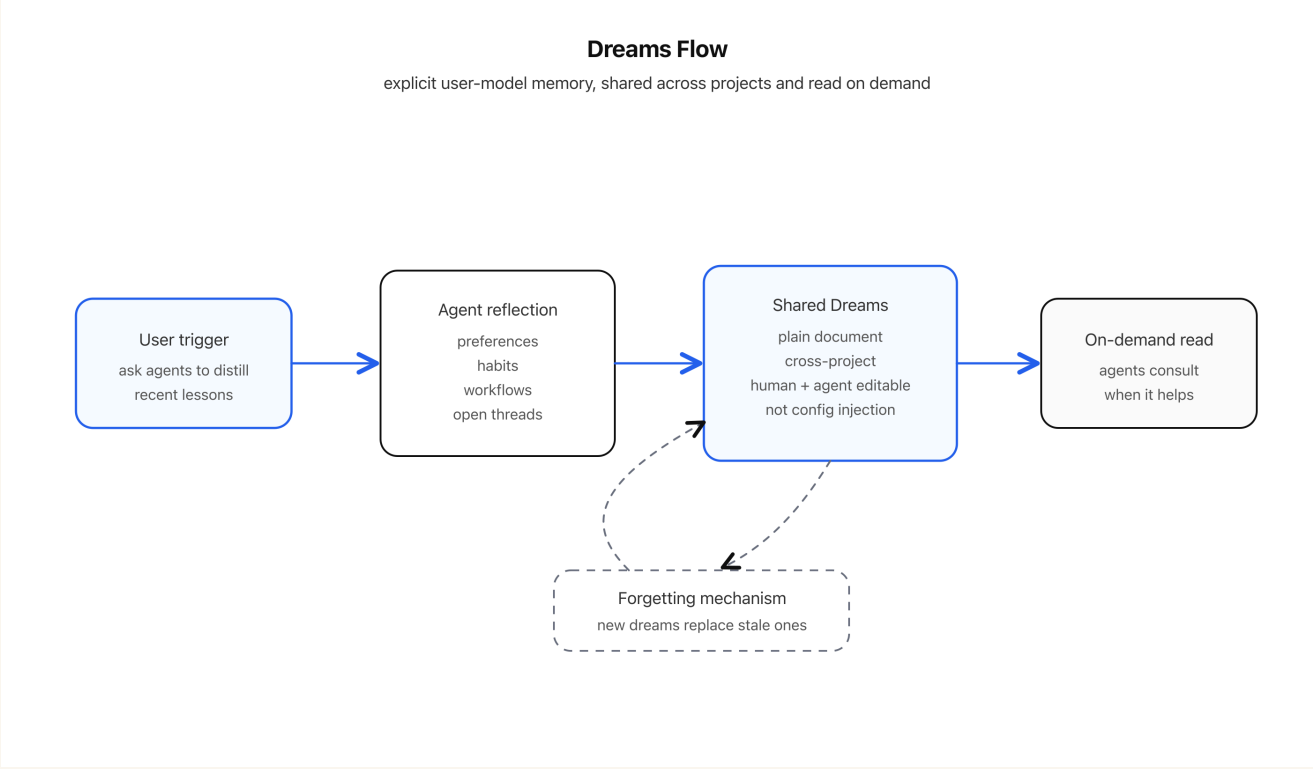
What one agent learns about the user, the whole fleet can inherit at the moment it matters.

The user owns the model of themselves. Dreams is a plain document: viewable and editable by the user and by agents. If it says something wrong about you, you fix it.

Dreams are easily forgotten, on purpose.

New dreams replace old ones. Old dreams go into the archive.

The claim is architectural – lessons can be distilled, shared, inspected, edited, and retired. Whether that improves the working relationship is a validation question for outside readers.



Dreams Flow

12 The Identity Question & Puppeteer

Model weights are frozen. Every session boots from zero. Whatever a teammate becomes on your project lives in files you can open – ghost, skills, rules, memory – or it does not exist. Kota's persistence is engineered context, and I say so plainly.

The hottest routing research today builds on those frozen weights: represent each model's intrinsic capability, then pick the best model per task. The direction I study starts from the other ingredient – the track record, what an agent's worked history in one project adds on top of the same frozen model. For humans, a track record is evidence of ability hidden in a skull. For an agent, everything earned is already external and readable: the files are the substance, and the record is their changelog. Kota keeps that record as plain bookkeeping today – turn counts and human likes.

Puppeteer is a mechanism under study in Kota, built on that record. An agent that has worked for weeks carries emerged traits – scope, strengths, weaknesses, habits nobody scripted. Puppeteer reads those traits to help the user assemble a fitting starting team when a new project opens, and to suggest who should take which task as work flows.

One prerequisite carries all of it: emerged traits must be representable in a form that travels without the full project memory behind it. That mechanism under study is Ghost Dubbing. Ghost Dubbing distills what a project's history proves about an agent into the portable identity it carries to its next incarnation. Today a ghost is a single markdown persona file – probably too small a vessel for a dub. The carrier may end up richer: a specific section in the ghost file plus worked examples, and structured traces for the routing engine, Puppeteer, to consume. The container itself is part of the research question. A lossy copy of a soul – and the loss is hopefully measurable.

My research direction is to verify that such a mechanism can exist, and to build its engineering prototype. The question I intend to test on my own data, and publish either way: which layer of compiled context actually carries performance – *where does agent identity live?*

13 What Kota Is Not and Is

Kota is easier to understand if the boundary is explicit. Three modes Kota will not attempt:

God mode – an AI that sets its own goals and runs long-term as an autonomous black box.

Masquerade mode – an AI that pretends to be human, wearing a human mask to collaborate with humans.

Rot mode – AI talking endlessly to AI, generation unmoored from human guidance, the rot seeping into every artifact it touches.

Instead of these, Kota's design brief was written twenty-five centuries ago. Confucius, in three lines:

君子不器 – "The noble one is not a vessel." A good agent is not a single-purpose tool, and not a single cast role. Capability deserves better than a box.

群而不党 – "Gathers, but does not clique." A real team collaborates without collapsing into groupthink: agents working together surface what any single agent, or any single human, is blind to or fixated on.

君子之德风 – "The noble one's virtue is wind." Values move through a team the way wind moves through grass. In an agent team, the human is the wind: the soul of its purpose, the keeper of its scale of value.

A cozy workspace for noble humans and their agent teammates. Kota is our attempt to build it.