Shinkai

Edward Alvarado
edward@dcspark.io
Shinkai

Nicolas Arqueros
nico@dcspark.io
Shinkai

Jay McCarthy
jay.mccarthy@gmail.com

Introduction

Shinkai is a novel deployment of modern AI technology that dramatically increases the utility of AI, similar to how spreadsheets transformed personal computing and displaced calculators. Shinkai relies on the convergence of novel AI planning, peer-to-peer decentralized communication & sharing, and a novel application of blockchain to achieve this.

AI tools today are too myopic in their applicability. Most evangelists believe that the next algorithm model update will produce 100x better results that avoid this myopia; but Shinkai believes a new structure to how AI is used, managed, and deployed will solve the myopia faster. Shinkai can make this more practical and deployable today with a thoughtful application of blockchain to build a peer-to-peer network of shared AI-backed services. This use of blockchain balances the many faceted problems of privacy, cost, and data integrity.

In this paper, we elaborate on all of these issues. We first discuss the current state of AI tools, then the specific problem that Shinkai solves, generally how Shinkai approaches its solution, how Shinkai uses blockchain, and finally delve deeper into some of the pertinent details about Shinkai's solution.

AI Tools Today

The modern era of AI started in late 2022 with the release of ChatGPT and since then AI tools have begun to slowly permeate the computational infrastructure. They are implemented in three main ways.

First, through ChatGPT's original interface of a dialog with an agent that expands on the Google search bar and has a running thread of conversation on a topic. Second, through a generalization of existing text completion systems, like predictive text and code completion, but expanded into domains where it was not previously used, like spreadsheets and design tools. Third, through behind-the-scenes improvement of existing applications, like better phone digital assistants, better data analysis summary, and better custom ad generation and placement.

Although interesting and impactful, these three primary approaches to AI remain shallow and fail to offer a truly transformative experience for humans. At Shinkai, we see this as being similar to the state of computing in the 1970s and 1980s. Back then, computers were mainly experienced in two ways: either as crude, one-size-fits-all calculators, or as custom-built systems tailored for very specific tasks.

With a calculator, a user could perform computations in a one-off and forgetful-manner that supplemented the modality where the "real work" was happening: the user needed a pad of paper on the side or a some other resources to plan how to use the calculator and how to integrate its results into the end product. This is analogous to how the AI chat dialog is separated from the larger endeavour of, for example, preparing a research report across many tools, from document editors, spreadsheets, YouTube and Internet research, diagram preparation, and so on. While the "generalized text completion" modality of working with AI alleviates this a little bit, it suffers because the AI interaction does not transcend and integrate multiple tools—it is as if every application had a bundled calculator for doing calculations conveniently within that application.

On the other hand, in early computing, a point-of-sale (POS) terminal or inventory tracking system was designed for one specific purpose: helping with sales or tracking products. These systems were

built in a way that made using them feel like a completely different experience from using a simple calculator. The cashier didn't need to think about how the computer was doing calculations—it was all hidden behind the scenes. These kinds of systems were built using specific programming tools, like assembly language or BASIC, which exposed the underlying power of the calculator to experts. It was standard practice in the first era of computing to buy a computer expecting only to receive a programming manual and a compiler, and needing to build your own application by-hand for your uses. This is analogous to the way that AIs are being used in the background by expert programmers to make some applications better and more performant.

In computing history, neither the calculator nor the bespoke application set off the personal computing revolution, instead it was VisiCalc: the first spreadsheet editor. A spreadsheet editor is a tailored application that is reprogrammable at a much higher-level than assembly or programming languages, but is an intuitive generalization from the calculator. With the advent of the spreadsheet, normal smart people could build spreadsheets for their own businesses and personal endeavors. These spreadsheets were like long, interconnected calculator sessions that were repeatable, shareable, and could be filled with new values and datasets depending on changing conditions and scenarios. Famously, entire industries like finance and accounting are maintained by staggering towers of Excel spreadsheets.

This history is striking to a modern programmer, because every spreadsheet is a trivial 20-line Python program. Python isn't that different from BASIC: why didn't everyone in the 1980s just learn BASIC and write their programs in BASIC instead of VisiCalc? Is it just that they didn't have VSCode with Github CoPilot and modern Python tutorials to teach them how to do it?

Shinkai's position is that a spreadsheet is actually better than BASIC or a calculator, because it enables everyday users to build and adapt complex workflows in a familiar, intuitive environment—without deep programming expertise. Shinkai's position is that the world of AI today needs its own "spreadsheet"—an accessible, flexible platform that democratizes powerful AI capabilities and invites anyone to create, share, and evolve new solutions.

The Problem

Current AIs are excellent at answering pointed questions, summarizing text, or giving a few small suggestions, but like calculators, they fail when a user needs to do more than a small task. We want our AIs to be genuinely helpful and capable of taking action on our behalf, without requiring us to guide their every move. We want them to expand their data set and capabilities based on the problem we present to them, as well as our personal data set, rather than only what they were trained with. Shinkai is not unique in recognizing this basic problem of modern AIs. Large and well-funded teams like OpenAI and Amazon are trying to solve it, but with a different approach.

OpenAI accepts the current paradigm and tries to maximize the performance of the "calculator"-style interaction through new models, such as `o1`, while providing the infrastructure for developers to build bespoke applications using AI. In this way, OpenAI is like Texas Instruments, which produced an excellent calculator and programming language, TI-BASIC. Their bet is that there will be a breakthrough comparable to the original GPT that will make the "calculator" interface 100x better, and structural improvement will be unnecessary. Even systems like Operator only incrementally improve this calculator by allowing it to parse a few screenshots and press a few buttons on a Web page.

On the other hand, Amazon Alexa is building the first version of Windows 1.0: a suite of AI-powered tools that is good enough for everything that the typical customer wants to do, but utterly useless when it goes beyond those slim uses. Their audience is wider: mass-market consumers that want simple AI automation and are impressed with what works, rather than disappointed at what doesn't work. This strategy allows Alexa to spend more resources making its small domain work well, but isn't scalable.

OpenAI and Amazon recognize that for AIs to be truly useful, they need to interact with external APIs, and generally be capable of building and using repeatable workflows from within the AI, rather than only as a supplemental guide to a human performing a task. OpenAI believes one model can interact with any API and perform any action with enough reasoning capabilities. Amazon believes that they can hand-code enough APIs to do what consumers want. Shinkai believes in a third, better way.

Shinkai's Solution

Shinkai will solve the problem of and achieve useful AI with three general approaches: first, guided AI planning and interactivity to turn linear agent interactions into structured collaborations between humans and agents; second, a peer-to-peer network of shared agent workflows and services; and finally, a local-first agent infrastructure to use AI without compromising privacy, security, or cost.

Shinkai Planner. One of Shinkai's main technical innovations is a guided AI planner that structures an interactive collaboration between humans and LLMs. Typically users ask questions to AIs in a linear fashion, gradually improving the AI's understanding of the problem, and massaging the result until a satisfactory answer is achieved, while carefully avoiding going over the context limit. Sometimes an AI's context tracking fails and a user needs to explicitly bring back into focus old issues by injecting them into the next prompt. Sometimes an AI's default training set needs to be augmented with user-specified databases that are expensive to curate and connect. This structure is similar to how calculators that do linear work are managed by a set of research notes, databases, and related computations that must be stored outside of the calculator itself. Spreadsheets, at their essence, are not infinite tables of cells, but rather structures that capture and codify the typical use patterns of calculators in large projects: connecting related computations and embedding databases in sheets connected on the sides of the main spreadsheet computation. Shinkai provides the same style of structuring, but for AIs, by introducing a layer of abstraction over what the user wants from an AI. At a high-level, this structure is that Shinkai manages an interactively growing database of goals of the form "Given X, produce Y" where "X" is some input and "Y" is some goal. For example, a translation task might start with "Given 'Hello, how are you?', produce a Spanish translation" and then be solved by a translation workflow of the form "For all English X, Given X, produce a Spanish translation", which is solved by a one-shot interaction with an agent and a particular prompt, like "You are a translator from English to Spanish. Please translate 'Hello, how are you?' ". We might depict this structure as follows:

- User Task: "Given 'Hello, how are you?', produce a Spanish translation"
  - Workflow Template: "For all English X, Given X, produce a Spanish translation" with "X = 'Hello, how are you?' "
    - Workflow Solution:
      - Agent with system prompt "You are a translator from English to Spanish." and user prompt "Hello, how are you?"

Shinkai manages simple workflows and structures like this, as well as those that are more complex, including recursive, iterative, non-deterministic, and potentially failing structures. The most interesting aspect of this process is that the large (and potentially infinite) tree of goal interactions may require more input from users, which Shinkai reifies as a linear interaction where a deep sub-process inside of the agent requests more clarity from users to reach a final goal solution. This management extends to specifying the mechanism by which sub-agents are connected to larger agents and workflows with standardized and safe data sharing & coordination.

Shinkai Network. Unlike the static and siloed world of spreadsheets, or the isolated world of short AI chats, Shinkai's solution tree is shared, global, and peer-to-peer. In particular, goals, solution workflows, agents, and task solutions can be exposed publicly with varying levels of disclosure. Furthermore, goals such as "For all English X, Given X, produce a Spanish translation" are admissible, so new workflows are often the result of Shinkai interactions, which can then be shared (and improved!) with more Shinkai users. For example, in the Spanish translation scenario, the user task might be given the disclosure level of "Public, Anonymous" and posted to the network. The Shinkai Network manages solution assignment, bidding, vetting, and may result in a variety of solutions between expensive human translators, expensive AI models with low latency, and cheap AI models with high latency. While a translation task is fairly shallow, many tasks are larger and will be naturally divided into many small sub-tasks that can be solved in parallel by vastly different workflows, often resulting in new workflows that will be useful in the future. We imagine an "AI AppStore" where anyone can design and list agents & workflows for a variety of problems in an open and decentralized way, akin to how anyone can produce a novel interface for existing smart contracts in the wider blockchain economy, as we observe in systems like smart order routers.

Shinkai Node. Finally, Shinkai provides a local-first agent infrastructure where individual participants can run their own network nodes to provide and access Shinkai services. At a gross level, this means providing a convenient no-setup way for normal users to run a local LLM, query it, and allow others to query it. At a more advanced level, this includes a sophisticated private data management layer and mechanism to run computations on encrypted data, while pooling unused computational resources from CPUs to GPUs. This node infrastructure ensures that Shinkai Network's disclosure rules can always be enforced without forcing exorbitant costs, as would be the case if all computations had to run on expensive cloud resources, or forcing exorbitantly bad performance, as would be the case if all computations had to run on slow local resources.

This is just a brief overview of Shinkai's solution structure. We elaborate on each of these aspects in more detail in the "Architecture" sections below.

Shinkai and Blockchain

Shinkai Network is enabled by modern blockchain technology from stable-coins, to zero-knowledge computation, to privacy-preserving cryptography. Although blockchain is used by Shinkai in many ways, we focus on three in this document: basic stable-coin enabled micro-transactions; a staking-over-subscription service model; and a facility for bridging to existing high-cost services.

When a user posts a goal to the Shinkai Network, they can pay for that goal's solution. When a user discovers a useful workflow or offers their node up for computation, they can charge for those uses. Each of these uses are likely to cost extremely low amounts of money that are impractical for existing payment infrastructure to handle. This is a perfect use-case for a low-cost blockchain network with settlement based on stablecoins. Shinkai Network is particularly suited for this, because it has almost no consensus state, lacking features like consensus data availability or smart contracts. Instead, it only facilitates a bare-minimum of collaboration.

However, even a low-cost blockchain may be too heavy for the tiny transactions in the Shinkai ecosystem we imagine. For example, it is unlikely that transaction costs can be so low as to satisfy efficiently charging $0.002 for a single interaction. Shinkai's solution for this is called Staking-over-Subscription. When users stake tokens in the network, they gain access to a stream of requests at no-cost, but their use is tracked and the resources they consume are compensated on a regular basis as value enters the network (such as when tokens are sold externally). This form of staking applies not only to tokens, but also to resources: a user can "stake" their node to be available for a certain amount of requests over a period to accrue tokens that can be spent in the same period, saved for future periods, or cashed out as an income source. (Figure 1 shows the protocol flow of paying for a service using Shinkai.)

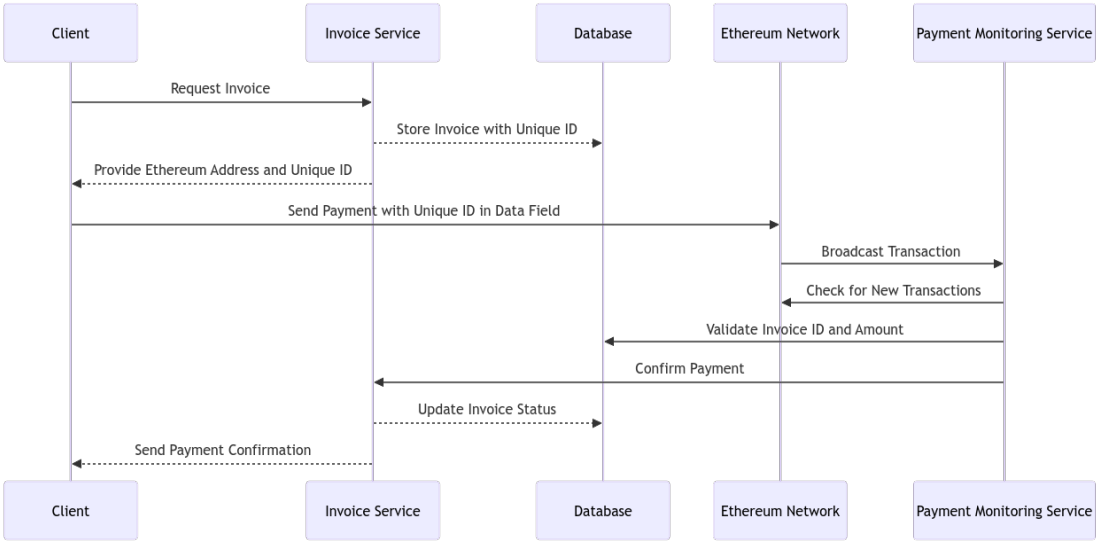


Figure 1: Shinkai Invoice Message Flow

Finally, Shinkai helps users stake more subtle resources, like existing subscriptions and access to non-Shinkai resources. For example, if you buy $5 of OpenAI credits that expire in a year, but only use $0.10 a month, then you can resell the remaining credits and amortize the cost of the many small transactions that other users would have had to buy those same credits. With expiring and monthly resources like this, Shinkai can help users who want to buy such resources find cheap ones (such as ones that are about to expire and are therefore being sold at a severe discount). Shinkai helps both sides of these transactions: users who have resources to be sold, and users that are on the buying side.

Architecture: Shinkai Planner

In this section we elaborate on one aspect of Shinkai's architecture: Shinkai Planner.

Shinkai Planner represents goals as desired productions of output from available input knowledge. As it plans a solution to a goal, it may synthesize new intermediate knowledge and intermediate goals. This process is represented by a logical deduction system akin to traditional AI systems. Let us elaborate.

A proposition of the form "A x B x C $\rightarrow$ D" means "Given an A, B, and C, we can produce a D". For example, "Pp : Given a radio and a transmitter, we can produce a phone" is a proposition like this. Some propositions have the same conclusions (D) with different antecedents (A, B, and C). For example, "Ps : Given a number, we can produce a string" and "Pb : Given a boolean, we can produce a string" are two propositions with the same conclusion, but different antecedents (and presumably different meanings behind the produced strings). Early AI systems worked by "backward chaining" that operated from a goal (D) and a database of propositions (Pp, Ps, Pb) and built a tree of possible ways to derive the goal, by choosing propositions until the search resulted in propositions that were satisfiable without antecedents. This old "logic"-style of AI was considered a failure by the mid-1980s, because it was cumbersome to design large databases of propositions that could derive all desirable human-level behaviors. Instead, modern "machine-learning"-style of AI was developed which is based on predicting function outputs from "small" (actually large) sets of example input/output pairs.

Shinkai's Planner is modeled on this style of "goal resolution" system, but where the propositions, (called "solvers") are drawn from three categories:

1. Deciders — These are algorithms that can always produce a solution from input. For example, normal computations like addition, JSON parsing, and database operations are deciders. These are assumed to be fast, cheap, and reliable, but uncommon: most problems do not have deciders.
2. Oracles — These are humans that can be asked to produce a solution from input. For example, we can ask the user if a test case meets their expectation of the behavior of a function. These are assumed to be slow (because the user has to think), expensive (because the human is impatient and doesn't want to answer lots of questions), and are risky to use, because we typically trust their output without being able to verify it.
3. Agents — These are AIs that can be asked to produce a solution from input. For example, we can ask the agent to generate a TypeScript type that corresponds to a plain-text description of a value. These are assumed to be slow (because the AI is far away), mildly expensive (because the AI is costly to interact with), untrustworthy (because the AI is easily confused and hallucinates), but have the advantage of solving many problems that deciders cannot.

Given a goal, we can select a solver that can solve goals of that form. We always prefer Deciders, then Agents, then Oracles. Each solver can propose new goals (i.e. request more input from what is available), solve the goal internally, or fail. If a solver fails, then we can ignore that solver (or try again), and return to earlier decision points and try a different solver. For example, we may give up on a Decider and switch to an Agent, but when that fails we may fall back to an Oracle. During the operation of the resolution system, we maintain a tree of the interconnected relationship of goals and solvers. Whenever we discover a failure, we try to optimize to minimize Oracle interaction. This means that we may explore many possible different Deciders at multiple levels of the tree before asking an Oracle (human) for advice, because we assume that interaction with them is scarce.

Let's work through a concrete example using a hypothetical database of solvers. Consider the goal, "Write a function that returns the average length of sentences in a document". There are no obvious Deciders for this task (but an analogous problem might have a Decider if a Shinkai user has previously requested a previous solution to this problem.) There is an obvious Oracle: we can ask a human to write the function, but this is a last resort, because we assume that most users can't write programs like this, or at least the ones that can are expensive. But, there are many possible Agents. One Agent ("Direct") receives the entire goal and directly produces a function. How does the resolution engine know that the function is correct? It could just trust the Agent or it could generate a new goal "Check if this function that returns the average length of sentences in a document is correct", where we have the opportunity to ask an Oracle if it looks correct. Given that we expect this new sub-goal after using the "Direct" Agent, it may be more productive to break the task into smaller sub-goals:

1. "Given a function purpose, what should the input and output be?"
    - This goal would take the original goal, "Write a function that returns the average length of sentences in a document", and derive an input "A document" and an output "The average length".
2. "Given a data description, what should the type be?"
    - This goal would take the data description "A document" and derive a type such as "string".
3. "Given a function purpose and types for the input and output, derive a test case"
    - This would take as input the goal "Write a function that returns the average length of sentences in a document", and the types "A document as a string" and "The average length as a number", and derive test cases like "("This is a test. This is not.", 3.5)".
4. "Given a function purpose and types for the input and output, and test cases, generate a function body"
    - This is like the "Direct" agent, but has far more context available
5. "Given a function and test cases, check if the function satisfies the test cases"
    - This could be implemented with a Decider, but perhaps an Agent could be useful in some cases (such as when the actual output is "semantically close" but not identical to the expected output; for example, "toSpanish("Hello") = "Diga" but expected "Hola" ".

In each case, these sub-goals could be solved by their own independent Solvers, perhaps Deciders, Agents, or Oracles. This particular set of sub-goals (and their interconnected nature) is just one example of a possible "plan" to solve the problem and in fact represents a "Decider", which algorithmically composes solutions to sub-goals to construct a final overall solution.

If we zoom in on step four, where given more context than just the plain-text description of a goal, we can see that there are many possible ways to write a function. For example, the function could work by taking the input and splitting it into a list of objects, analyzing them separately, and then combining the output as `(input: X) => split(input).map(process).then(combine)` where `split`, `process` and `combine` are three other functions. This is a plausible strategy for our sentence length problem, because `split` could be "Break a document into a list of sentences", `process` could be "Count the length of a sentence", and `combine` could be "Average a list of numbers". We could try to use a "Direct" Decider authoring approach that discovers this function or we could implement a "List Processor" Decider that can solve any input problem if it can determine what the purposes of `split`, `process` and `combine` are and generate their implementations (with tests cases validating those implementations).

If we step back from step four, we must decide how to decide if the "Direct" or the "List Processor" Decider should be used. We could use a general purpose "Try many options" Decider, or we could use a "Decide the best strategy" Agent, or we could use a "Ask the user which strategy to use" Oracle. Thus, the resolution tree is itself the subject of the resolution process.

Shinkai follows the typical AI interaction loop of a linear sequence of Oracle interactions as the basis of the user interface, but maintains the tree structure which is hidden from the user. We anticipate that advanced users may be able to directly read the tree with highlighting to indicate which style of Solver

was used at each point in the tree and offer the user the choice to override and influence any Solver by acting as an Oracle for that particular goal. Our hypothesis is that a user experience like this will be understandable and powerful for normal users in the same way that the spreadsheet abstraction is powerful. Despite this particular hypothesis about a UI, the linear sequence model is highly usable and shows that this model is expressive and useful.

Shinkai Planner is a novel and promising way of interacting with AIs for complex tasks. Although in our examples, the user's goal is explicitly to write a program, in practice we expect that most users will have object-level goals, which Planners will turn into function-level goals. For example, "What is the average number of words in each sentence of the Bible?" is unlikely to have an existing Decider, be feasible for a human Oracle, or feasible for a trivial LLM-based Agent to immediately answer; instead, we expect that most goals will be initially picked up by a Decider that transforms the goal into, "What function would solve this specific problem?" and then derive that particular function.

Despite its novelty, Shinkai Planner is a combination of the best parts of old logic-style AI, extended with the capabilities of modern AI. We anticipate that the space of possible Solvers is huge, including categories like "A function that uses a database" where the Decider walks through the process of deciding the table structure, the queries, and populating the database. We anticipate that the ecosystem of Shinkai users will produce and share Deciders, Agents, and Oracular answers collaboratively in a way that is analogous to how on traditional computing systems there are many programs, many files, and many network services. We think that there is potential that this abstraction could be the basis of a totally new way of interacting with computational resources.

Appendix 1 shows a simple example of a linear Planner interaction, annotated with commentary.

Architecture: Shinkai Network

In this section we elaborate on one aspect of Shinkai's architecture: Shinkai Network.

The Shinkai Network is the decentralized peer-to-peer network that facilitates real-time sharing of AI data (i.e. embeddings from the latest Web content, as well as personalized data from users) and enables AI agents and users to message each other. The network is integrated directly into blockchain to implement a decentralized and censorship-resistant system for Shinkai Identities. This enables nodes to discover each other and communicate autonomously, without any need for middlemen. In this section, we talk about the basics of the token and network, how identities are registered, and how network messages are exchanged.

At the heart of the network is the KAI token. It powers the identity registration system through staking and powers the peer-to-peer network by being a scarce resource that prevents Sybil-attacks and other overuse issues. It also provides a decentralized funding mechanism whereby developers can receive and be staked KAI tokens to fund the development and operation of Shinkai agents and other smart contracts. Other than the messaging layers that we will discuss shortly, the Shinkai network is overtly designed to be a simple and straight-forward blockchain without any novel consensus algorithms. This is a strategy to de-risk the platform and limit the amount of innovation to only the parts that are crucial to Shinkai's operation. The most interesting thing about the network is the way the decentralized blockchain is used to bootstrap peer-to-peer direct, encrypted transfers of large (& non-public) workloads and datasets, as discussed below.

Shinkai Network allows users to mint identities by staking KAI tokens to them. Identities are hierarchical indices that allow users to name resources (like agents, devices, files, data sets, and so on) and delegate the management of those resources to others. Top-level identities are expensive by virtue of an inverse relationship between the staking cost of an identity and the length of the identity. This ensures that desirable identity real estate (like `@@Auction.shinkai`) is more expensive than complex and low-value identities (like `@@Jay_and_his_pet_dogs_favorite_treat_to_eat.shinkai`). Identities are NFT-like in that they are mintable, burnable, and transferable independent of the initial producer without requiring complex cryptography or sharing of keys to facilitate such operations. Finally, the identity registry uses the staking mechanism to allow identities to publicly re-stake, or delegate, their KAI tokens to other identities, which is the means that services tied to those services manage and pay for subscriptions or other use.

Shinkai's peer-to-peer network uses a novel messaging layer to turn the public blockchain messaging and identity registry into a direct communication network for requesting work, data, and services. Suppose that two identities (perhaps, one human and one agent) need to communicate. The initiator looks up the receiver in the identity registry and learns its position in a distributed hash table (DHT) and its public key. The initiator can now create and route a message to the receiver requesting an efficient direct connection (such as with an IP address) and an efficient symmetric encryption key. The receiver can respond and then continue the interaction in a more efficient manner. The receiver can opt-in to how this is managed. For example, some receivers may run hosted servers that relay to their phones; while other receivers will refuse to ever share their direct connection information. In any case, the two parties can then exchange secret, encrypted information backed by a blockchain-authenticated identity and public-key-encryption pair.

Appendix 2 shows an example of this connection establishment protocol in action.

Architecture: Shinkai Node

In this section we elaborate on one aspect of Shinkai's architecture: Shinkai Node.

The Shinkai Node is the personal gateway for users within the network. These nodes serve as both consumers and providers of data, contributing to a decentralized environment where collective AI intelligence is continuously enhanced through shared data and resources. Users can install a node on their local machine with two clicks. Once installed, a node provides three main features: management of agents, management of data, and management of tools. We discuss these in turn.

First, Shinkai Node simplifies the installation and management of many different AI models. Like Ollama, Shinkai facilitates giving users access to LLMs, but goes further by providing a uniform interface and managing multiple long-term interactions with those models. While Ollama may provide access to "models", i.e. the raw prediction algorithms, Shinkai provides access to "agent", i.e. those models, but tailored for particular problems, domains, and data sets. This is done with an open source and plugin-style architecture that allows an ecosystem to grow around Shinkai, but without compromising data integrity and privacy standards that the blockchain community has come to expect.

Second, Shinkai Node manages data on behalf of users and provides that data safely to agents. There are two relevant aspects of this: first, the private data availability layer that allows nodes to provide data to other nodes and network-integrated service without completely revealing the data; and second, a novel vector-based file system for exposing data directly to agents. The data availability layer was discussed in more detail in our explanation of the network in the "Architecture: Shinkai Network" section, so we discuss the vector file system here. The easiest way to integrate data with an AI is to have that AI produce a vector embedding of its "parsing" of that data for later retrieval and use in specific queries. This process can be cumbersome to manage and it must be refreshed when models or data change. Existing databases and file systems are not designed around this use case. Thus, we designed a new user-space file system with exactly those properties: a hierarchical layout of original sources with vector embeddings of their contents, metadata, and summaries, such that each level of the hierarchy (including individual files) can be transparently made available to agents as needed. The node allows users to set a policy for each layer of the hierarchy, as well as for embeddings themselves, as to how that data should be shared and made available. These policies and layers are backed by Merkle trees that assert the identity and integrity of files, aiding in efficient sharing and synchronization.

Finally, Shinkai Node manages the creation of and interaction with containerized, AI-connected tools: scripts and workflows that solve problems for users. For example, if the Planner produces a translation tool, then this will be presented as a tool to Shinkai users within Node. In practice, this will be a Javascript program run by the local user with access to the correct resources, such as files and agent connections. It is not feasible for most users to safely run such programs, just like it is not feasible for them to set up Ollama and use LLMs directly. Shinkai provides a solution for this problem with its safe execution environment. The same policies that share data with external parties allow Shinkai users to share data with tools, both during creation, training, and operation of those tools. This aspect of Shinkai is analogous to a Web browser, which allows anonymous and untrusted programs (HTML, CSS, and Javascript) to be downloaded and run on users' machines, while giving controlled access to the

users' private resources (screens, devices, files, and so on). The most difficult and interesting part of this problem is to allow multiple tools with different implementations and trust levels to share information without compromising the privacy and integrity of that data. This sharing is most troubling in the recursive cases where an agent interacts with another agent to satisfy requirements, rather than having only hard-coded business logic internally that could be checked itself. We call this containerized system the Transient Execution Framework and discuss it in more detail in the next section.

Appendix 3 shows some concrete examples of the tool interactions enabled by Shinkai Node.

Shinkai Transient Execution Framework

The Transient Execution Framework securely runs small, standalone code snippets in a controlled environment. This approach addresses common pitfalls of naive code-execution designs—where code either runs unguarded in the user's main environment or requires extensive custom setup for each invocation—and it furnishes several key advantages:

1. **Automatic Isolation.** Each snippet is placed in its own temporary folder, complete with a dedicated cache, logs, and an optional "home" directory for internal file writes. This prevents cross-contamination of data or dependencies, which often occurs in naive environments where scripts share global installs or write to the same filesystem paths.

2. **Selective Resource Exposure.** Snippets can only access the resources the user explicitly grants.

   - *Local Tools:* The snippet may call containerized or system-wide executables, but only those whitelisted by the Node.
   - *Memory & Database Access*: If the snippet needs in-memory data, or permission to query a database, Shinkai Node ensures that only authorized tables or memory segments are revealed.
   - *API Keys & Credentials:* Sensitive tokens are injected on a need-to-know basis, preventing accidental leaks or broad misuse.

Shinkai Node integrates seamlessly with Docker for strong sandboxing when needed; otherwise, snippets run locally with restricted privileges. By contrast, naive approaches frequently run code with blanket access to system resources, risking accidental or malicious interference.

3. **Fine-Grained Caching Control.** Naive execution often re-installs dependencies for each run—or never cleans up after them. Shinkai's framework preserves cached libraries (e.g., Python wheels or Deno modules) between runs for performance, or resets them entirely to maintain a clean slate.

4. **Comprehensive Logging & Diagnostics.** During execution, every console message (stdout and stderr) is captured in a run-specific log. This ensures each snippet is fully auditable—no logs get lost in sprawling system output. Naive approaches often merge logs from different snippets or discard them, complicating debugging or usage analysis.

5. **Composability & Chaining.** Users can chain snippets to form more sophisticated workflows—forwarding outputs from one step into the next. In naive code-run setups, such chaining frequently requires ad hoc scripting to pass data around or coordinate multiple environment setups. Here, Shinkai Node orchestrates everything inside a unified transient framework, allowing ephemeral scripts to seamlessly hand off data or invoke external APIs.

Through these features, Shinkai's Transient Execution Framework preserves the convenience of quickly running user-supplied code, while also providing robust isolation, flexible caching, and carefully restricted resource usage. By managing database access, memory exposure, and tool availability, the framework secures each snippet's interactions with the system—ultimately reflecting Shinkai's broader philosophy of local-first, privacy-conscious AI orchestration.

Beyond handling single, ephemeral scripts, Shinkai's Transient Execution Framework accommodates more advanced scenarios—ranging from straightforward prompt chaining to fully autonomous, multi-step "agentic" solutions that involve iterative reasoning and flexible tool usage. Developers can create these solutions using external generators (like Devin or Cursor) or by leveraging Shinkai's own planner.

Regardless of which tooling you prefer, the Node's transient isolation and selective resource exposure unify everything under a common, secure execution layer:

1. **Multi-Step Prompt Chaining.** A complex process can be broken down into smaller, well-defined subtasks. Each subtask runs as its own ephemeral snippet, with intermediate results feeding the next step. This ensures clarity and maintainability: every snippet handles a smaller piece of logic, while isolation and caching are centrally managed.

2. **Parallelized Operations.** When multiple tasks can run concurrently—such as searching different databases or testing code fragments—Shinkai Node spins up separate parallel containers. Each container has its own resources and logs, so concurrency doesn't cause collisions or confusion.

3. **Orchestrated Workers.** Certain tasks don't neatly map to a predefined list of actions. Instead, a coordinating snippet can dynamically decide which sub-snippets to spawn based on partial outcomes or evolving goals. By combining ephemeral snippets with selective data-sharing, Shinkai Node ensures that each newly spawned snippet runs with only the resources and APIs it genuinely needs.

4. **Evaluator-Optimizer Loops.** Shinkai Node supports iterative improvement cycles, where one snippet proposes a draft solution while a second snippet evaluates or critiques it. Based on the evaluation, a refined attempt can be spawned. Because each iteration is ephemeral, every step is clearly logged and easy to review.

5. **Autonomous Agents.** For open-ended tasks where the number of steps can't be predetermined, an agent can repeatedly invoke ephemeral snippets that consult new data, retrieve partial results, and update its plan in real time. Crucially, Shinkai's selective resource exposure confines the agent's abilities, even when it's controlling powerful tools such as a Chrome browser.

- *Dynamic Browser Control:* Rather than hardcoding site-specific logic or building specialized crawlers, the agent spins up ephemeral snippets that drive a headless Chrome session. It can click links, fill out forms, parse returned data, and gather insights—adjusting its strategy on the fly.
- *Decision-Making on the Fly:* Because the agent reason about each step's outcome, it can spontaneously switch to structured workflow fragments if it deems them helpful (e.g., reusing a tested snippet for a particular subtask). This avoids rewriting code for each website or scenario, emphasizing reusability.
- *Guardrails & Logging:* Even with flexible, autonomous behavior, Shinkai enforces strict guardrails around data access, environment permissions, and usage limits. Detailed logs for each snippet let you trace exactly how the agent arrived at a decision.

By combining ephemeral snippet isolation with container orchestration, caching, and resource gating, Shinkai's Transient Execution Framework provides a robust foundation for everything from simple utility scripts to complex, AI-driven interactions with the wider web. Developers can move from single-run tasks toward sophisticated, agentic systems—knowing that every action is securely sandboxed, neatly logged, and controlled by the resource limits the user deliberately sets.

Shinkai Compared to Related Work

The myopia and limited scope of modern AI tools is widely recognized. Shinkai is not the only company trying to address these problems. But, our approach is unique and superior in many ways. Figure 2 presents a high-level comparison between Shinkai and competing frameworks.

As this comparison shows, there are a few common dimensions along which approaches are divided. For example, each approach is inherently based on a modular and collaborative architecture for agents„ but are divided on how to expose and use that architecture. Eliza, LangChain, CrewAI, and VirtualsAI are all environments that are "developer-first" frameworks that help developers create AI applications better; while Shinkai and OpenAI Operator are designed for users to interact with directly.

Most of the features presented in this table are unique to Shinkai: the transient execution framework, a native way to execute Python & TypeScript, a local-first execution model, and local-controlled data. The impact of local-controlled data on regulatory compliance is worth elaboration.

Feature / Framework	ai16z (Eliza)	LangChain	CrewAI	VirtualsAI	Shinkai	OpenAI Operator
Modular Agent Architecture	Yes	Yes	Yes	Yes	Yes	Yes
Requires Terminal/Manual Dependency Management	Yes	Yes	Yes	Yes	No	No
Requires Separate Coding for New Tools	Yes	Yes	Yes	Yes	No	Minimal*
Transient Execution Framework	No	No	No	No	Yes	No
Native Python Code Execution (Multi-version)	No	No	No	No	Yes	No
Native TypeScript Code Execution	No*	No	No	No	Yes	No
Blockchain Integration	Yes	No	No	Yes	Yes	No
Local-first Execution	No	No	No	No	Yes	No
User-Friendly UI (No Terminal Needed)	No	No	No	No	Yes	Yes
Team-based Multi-Agent Coordination	No	No	Yes	No	No	No
Privacy/Regulatory Friendly (Local-Controlled Data)	No	No	No	No	Yes	No

Figure 2: Shinkai Compared to Related Work

Shinkai's Potential Regulatory Advantage. Many privacy laws heavily regulate how user data is gathered, stored, or shared—creating particular risks for AI providers that rely on external servers. In those setups, user information may end up distributed across multiple data centers, making compliance with regulations like GDPR and HIPAA far more complex. Under HIPAA, for example, Protected Health Information (PHI) must be handled in accordance with privacy and security rules, and providers often need explicit Business Associate Agreements (BAAs) with AI vendors that process or store PHI.

By contrast, Shinkai's local-first design keeps sensitive data on their own machine. No global Shinkai server holds PHI or other regulated data. Users can decide exactly what, if anything, is shared, thereby reducing exposure to costly or complicated compliance issues. For instance, handling sensitive medical text locally bypasses the need to store such content in the cloud or trust an external party's HIPAA compliance, offering a more direct and cheaper path to meeting privacy requirements.

Although Shinkai's architecture does not eliminate all compliance obligations, it places data management firmly in the user's domain, potentially minimizing many of the risks that come with remote data processing. Ultimately, each organization should consult qualified legal counsel to ensure full compliance, but Shinkai's design offers a pragmatic way to sidestep major privacy and regulatory hurdles that often plague centralized AI solutions.

Conclusion

In computing history, everyone assumed that the future belonged to the hardware manufacturers that innovated a new chip every two years with better performance for less money. But software consumed the future through a killer product that made the toy of the personal computer into an economic workhorse that was essential for every business and, ultimately, essential for every human's daily grind. Today, everyone assumes that the future belongs to Nvidia's GPU manufacturing facilities or OpenAI's model training; but, the future belongs to Shinkai's superior way of giving normal people access to a more functional computation life, empowered by AI.

Appendix 1: Shinkai Planner Linear Interaction Example

Here is a simple example of a linear prototype interaction. We interleave user messages (noted with `User:`), system messages (noted with `System:`), hidden data manipulations (`Data:`), and comments only in the context of this document (`Note:`):

```
System: What is your goal?

User: Write a function that returns the average length of sentences in a document.

Data: Node 0 = N0.Goal = Function("Write a function that returns the average length
of sentences in a document.")
Data: N0.Solver = Decider(FunctionPlanner)
Data: N0.Child 0 = N0.C0.Goal = Spec("Write a function that returns the average
length of sentences in a document.")
Data: N0.C0.Solver = Decider(SpecPlanner)
Data: N0.C0.C0.Goal = IOSpec("Write a function that returns the average length of
sentences in a document.")
Data: N0.C0.C0.Solver = Agent(Specifier)
Data: . . . query LLM . . .
Data: N0.C0.C0.Answer = { input: "a document", output: "a number" }
Data: N0.C0.C1.Goal = TypeSpec({
        desc: "Write a function that returns the average length of sentences in a
document.",
        input: "a document",
        output: "a number"
    })
Data: N0.C0.C1.Solver = Agent(TypeSpecifier)
Data: . . . query LLM . . .
Data: N0.C0.C1.Answer = { input: ["string"], output: "number" }
Data: N0.C0.Answer = {
    desc: "Write a function that returns the average length of sentences in a document."
    input: "a document", input_type: ["string"],
    output: "a number", output_type: "number" }
Data: N0.C1.Goal = TestCases(N0.C0.Answer)
. . .
Data: N0.C1.Answer = [
  ("This is a test. This is not.", 3.5),
  ("This is not a test", 0),
  ("This is a bad test. Okay?", 4),
    ]
. . .
Data: N0.C2.Goal = FunctionBody(...)
. . .
Data: N0.C2.Solver = ListProcessorDecider
. . .
Data: N0.C2.C0.Answer = `(input: string) => input.split(".")`
```

```
. . .
Data: N0.C2.C1.Answer = `(input: string) => input.split(" ").length`
. . .
Data: N0.C2.C2.Answer = `(input: number[]) => input.sum() / input.length`
. . .
```

Note: Notice that a lot of work has gone on behind the scenes without the user having to give any instruction or interaction. A variety of agents and deciders have worked together with many many calls to LLMs behind the scenes, but we don't bother the user until we have a problem (or solution!).

Note: At this point, we have a complete program, but the test cases are wrong, so the resolution engine is going to fail, because N0's Solver will run the tests and then won't work. At this point, should it...

1. Try a different Combiner step in the ListProcessorDecider?
2. Try a different strategy instead of ListProcessorDecider?
3. Try different test cases?
4. Try different input and output types?
5. Try different input and output descriptions?

The correct thing to do is 3, but if we go to the "most recent" AI interaction, we would choose 1 and if we went to the oldest we would choose 5. It may be possible to have a way to "realize" that 3 is the most suspicious step, either by recording the confidence of each step or by using an Agent to select which step to review. But, in our prototype, we always go to the oldest Agent step and get confirmation from the user. [We do not go to the oldest Decider step, because we assume that Deciders are correct if they apply at all.]

```
System: Is "a document" a good input to this function?
User: Yes
Data: N0.C0.C0.C0.Trusted = True
```

Note: Because the user confirmed, the tree doesn't change, but we record our increased trust in the answer.

```
System: Is "a number" a good output to this function?
User: Yes
Data: N0.C0.C0.C1.Trusted = True
Data: N0.C0.C0.Trusted = True
```

Note: Because all of the node's inputs are trust, the decider becomes trusted and will no longer be a place where we can back-track to.

```
System: Does the type "string" represent "a document" well?
. . .
```

Note: This is an example of a confirmation that is a good candidate for an AI to answer. This is where the meta-nature of the tree appears in practice. Rather than actually mandating that the user be asked a question, we instead generate a goal of the form "Confirm N0.C0.C1" and one way to confirm it is with an Oracle, but another way is with an Agent.

```
. . .

System: Is ("This is not a test", 0) a good test case?
User: No, the answer should be 5
```

Note: At this point, the user doesn't confirm, but revises. The user output will go into a new goal of the form "Update test case ("This is not a test", 0) with the answer should be 5" and we will get a new test case ("This is not a test", 0). The whole tree will be re-evaluated and it is possible that new code will get generated (but probably not) and the whole test-suite will re-run. It still fails because the third test case is also wrong, but we don't re-confirm things that have since become trusted and we immediately ask

System: Is ("This is a bad test. Okay?", 4) a good test case?
User: No, the answer should be 3.

Note: Now, we update the test as before, re-run everything, and all the test cases pass.

System: Your function is

```
const averageLength = (input: string): number => {
 const split = (input: string) => input.split(".");
 const process = (input: string) => input.split(" ").length;
 const combine = (input: number[]) => input.sum() / input.length;
 return split(input).map(process).then(combine);
}
```

It passes these test cases:

```
("This is a test. This is not.", 3.5)
("This is not a test", 5)
("This is a bad test. Okay?", 3)
```

Would you like to generate any more test cases?

Note: Now the entire interaction is done. In the non-linear non-prototype user interface, we imagine that the tree will be exposed to the user, so they can see what decisions were being made and they will have the opportunity to influence any decisions even when they are not asked.

Appendix 2: Messaging Examples

Figure 3 shows an example of Shinkai message in action:

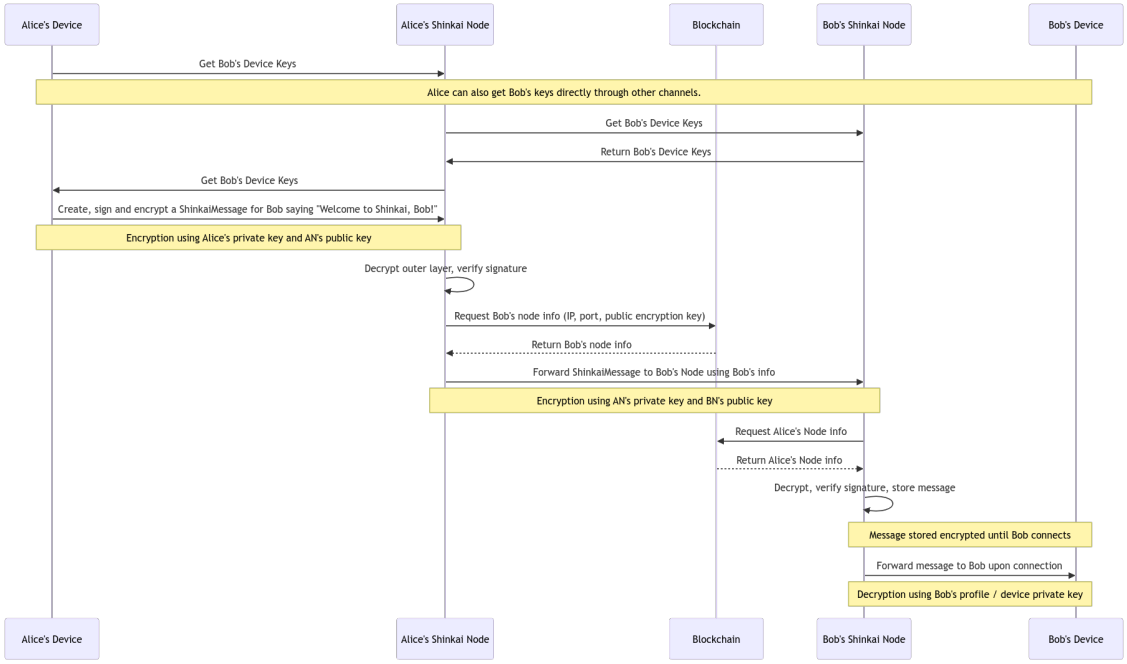


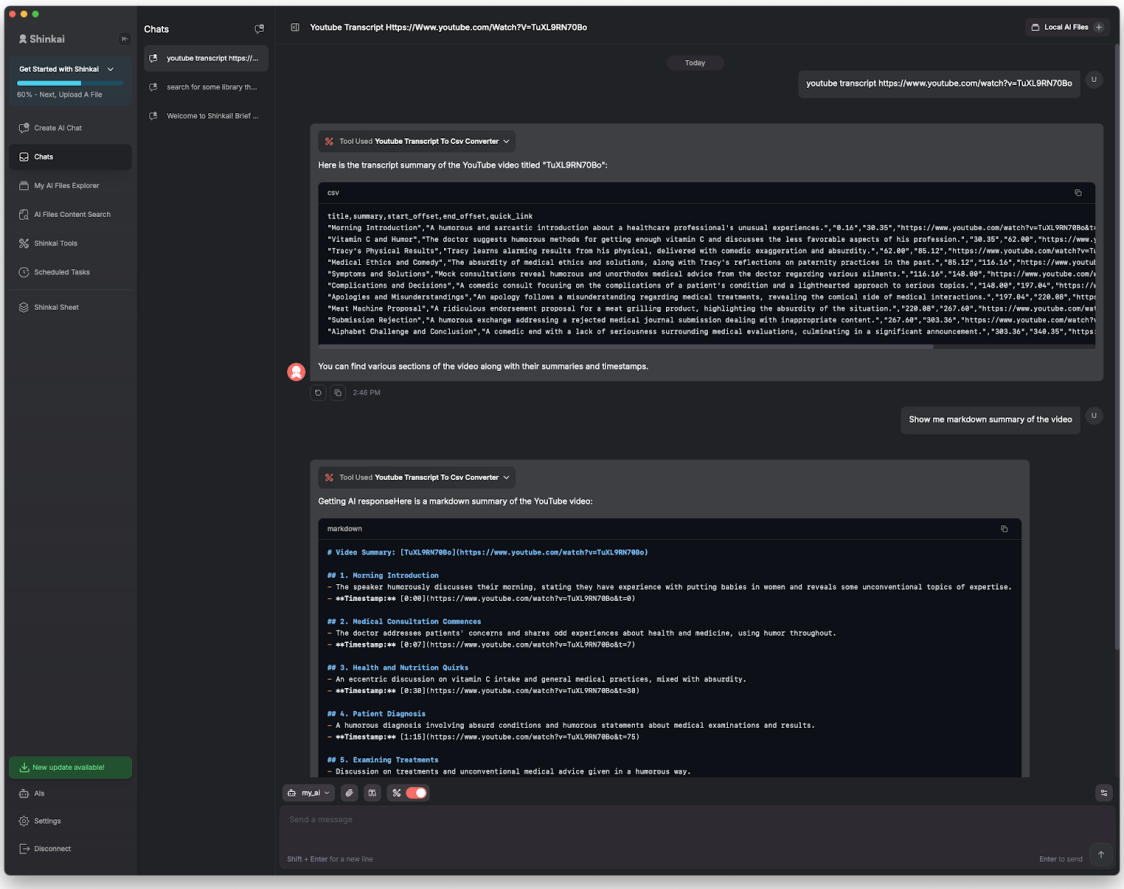
Figure 3: Shinkai Message Flow

Appendix 3: Example Shinkai Tools

This section gives three examples of Shinkai produced tools. For each tool, we show how the user interacts with the tool and we show how the user produced the tool with Shinkai.

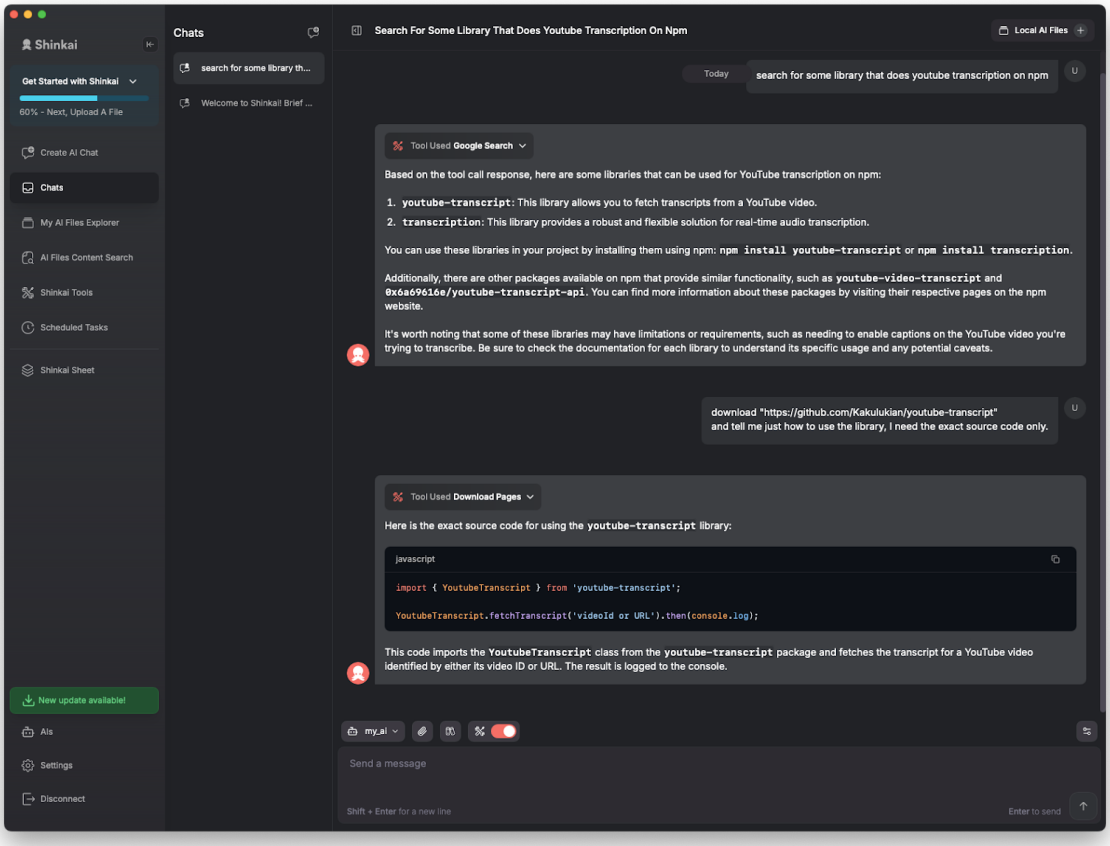
Basic Youtube Transcript Summarizer

This tool allows users to get a summary of a YouTube video by first having an agent read a transcription, then summarize it. This screenshot shows the user interacting with the tools by requesting a transcript and then a summary:

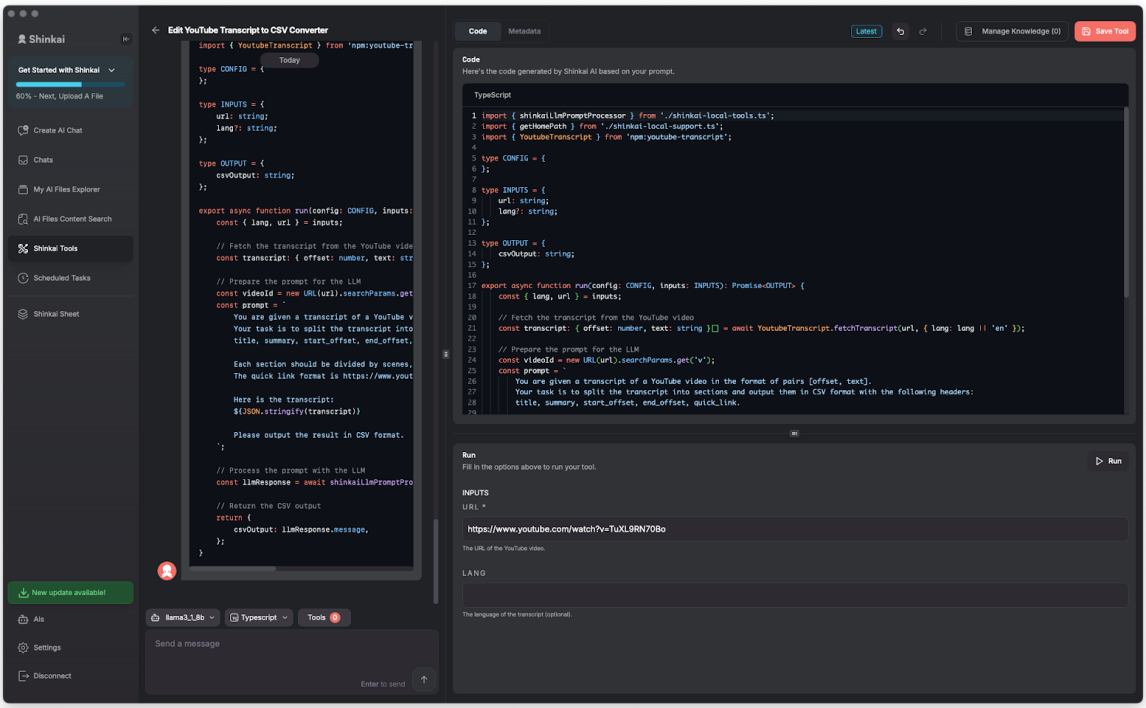


Although this looks almost exactly like a normal ChatGPT-style dialogue, behind the scenes Shinkai has mapped the request to a previously authored tool from its tool database. The tool was originally created by other interactions with Shinkai.

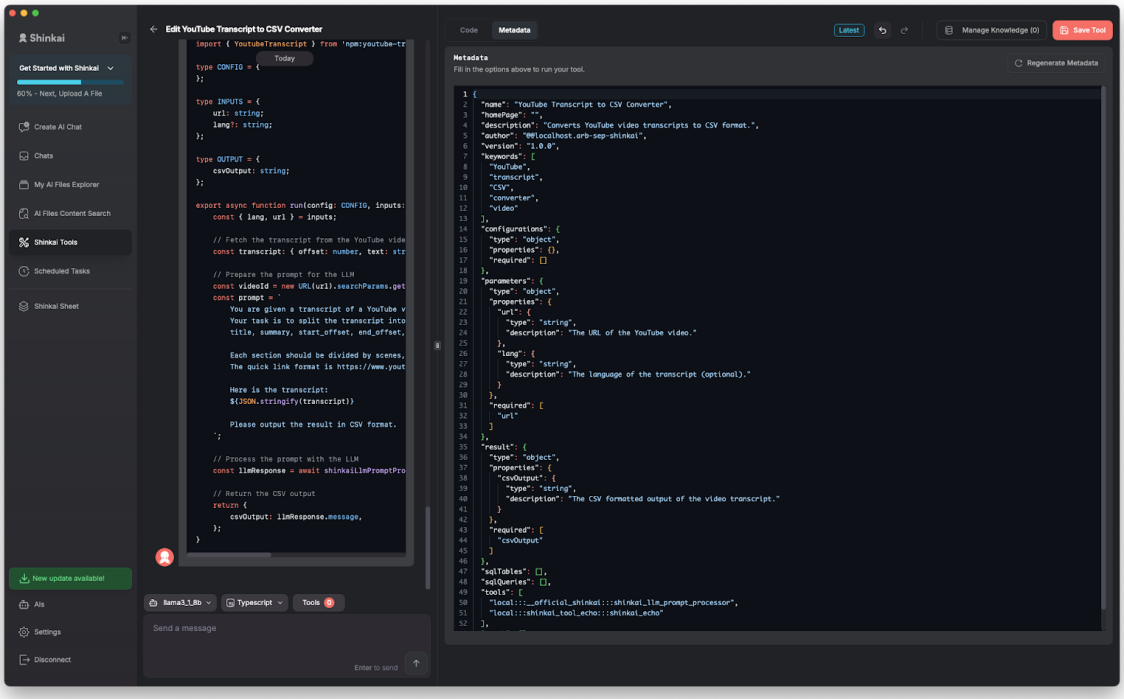
First, the user asked for a transcription library reference:



And then asked for the tool to be constructed using it:



That tool is then given metadata for it to be integrated into the tool database for later use:

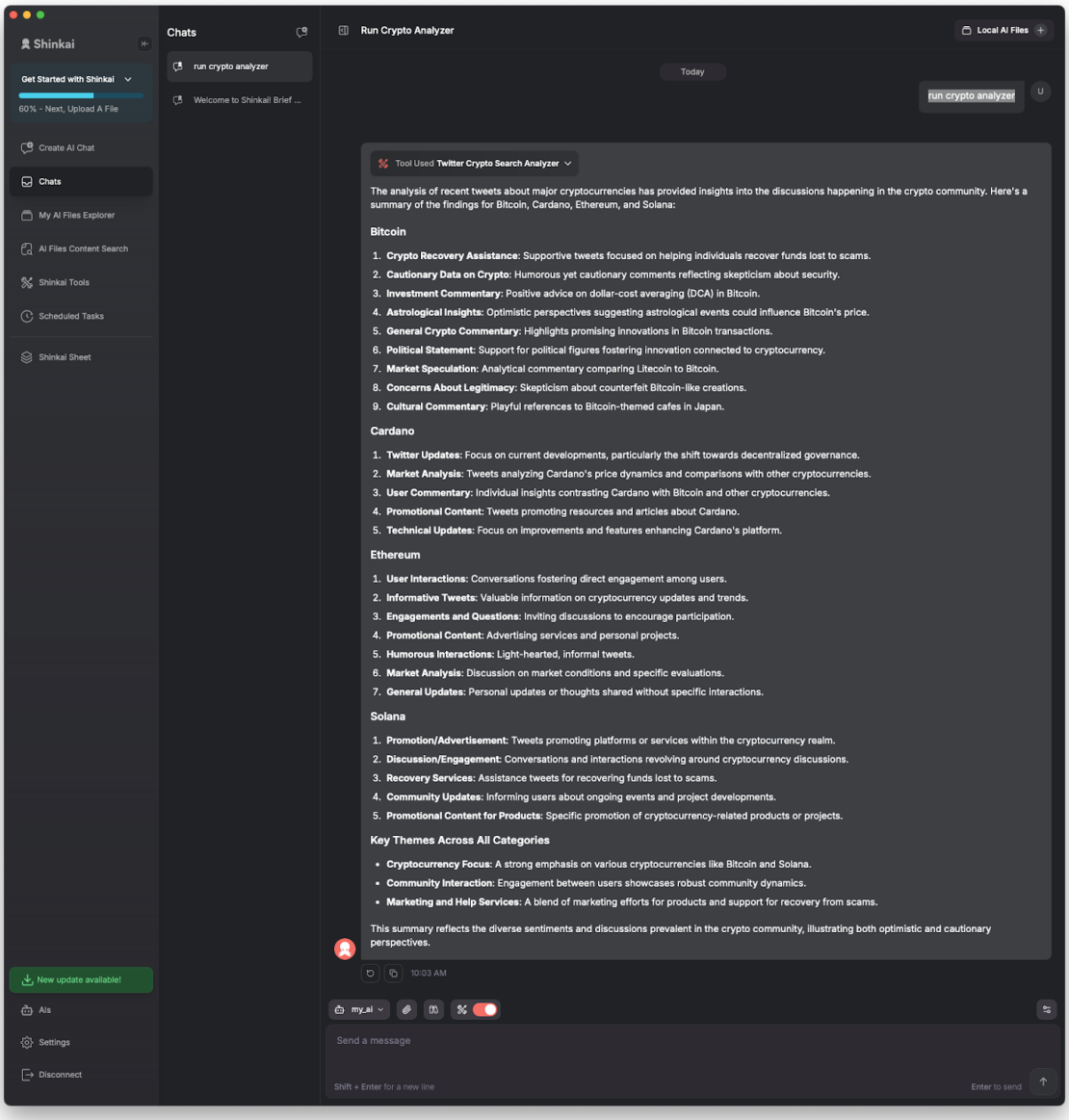


These examples show a basic use of Shinkai where the now-familiar dialogue-style agent interaction is enhanced by a powerful tool database and tool construction system.

Crypto Twitter Analyzer

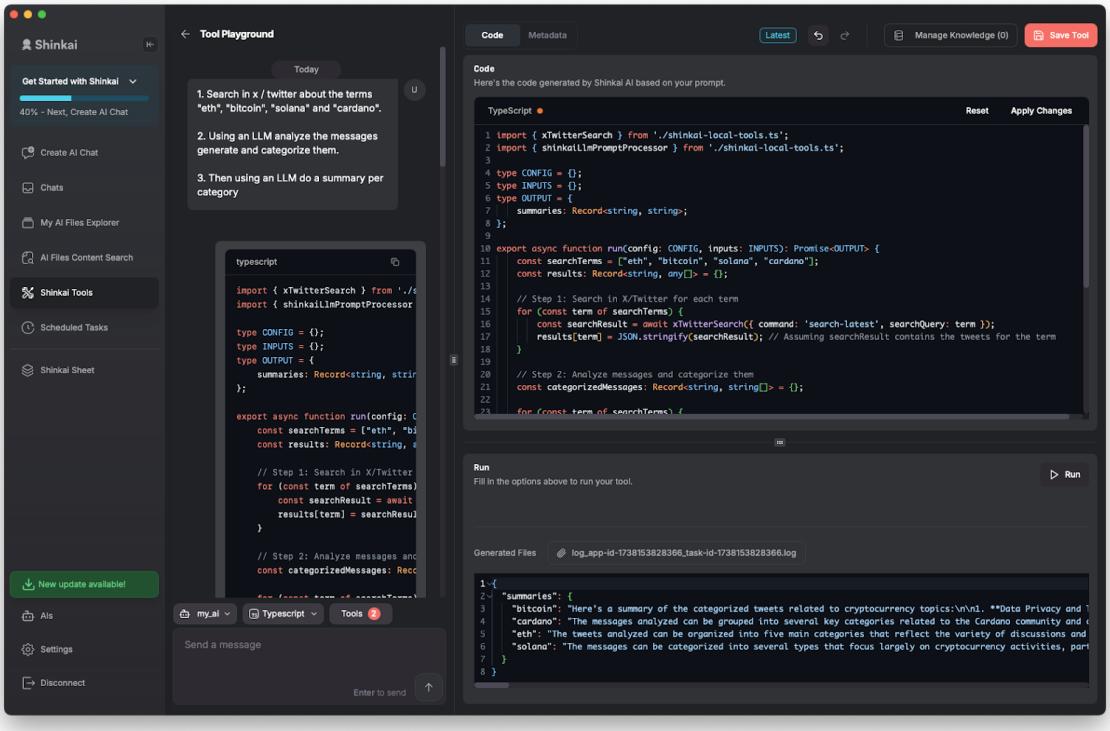
This tool gives a user a summary of the sentiment of the Twitter community on a number of crypto markets. It shows a nice blend of using complex APIs and searching, as it needs to get Twitter data; as well as a nice application of the natural language understanding that AIs have in its processing of that data. Finally, it shows a use case likely to appeal to a blockchain-engaged crypto user.

This first screenshot shows what it is like to use the tool:

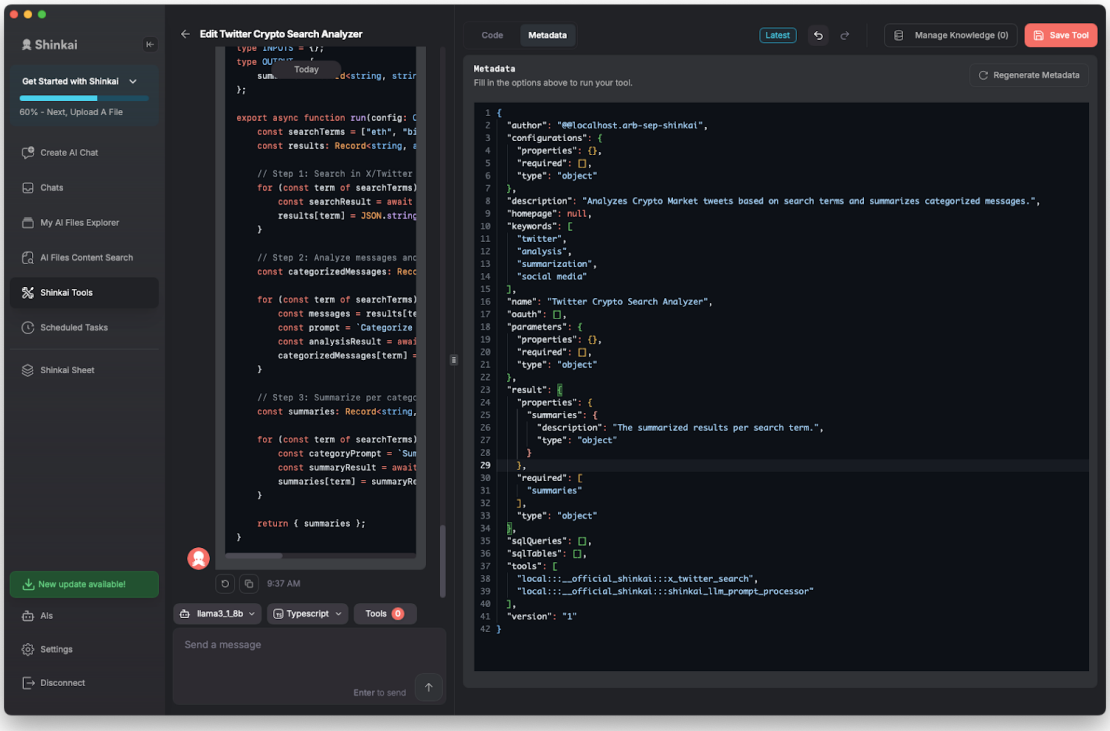


Next, we can see the code that Shinkai generated when it was requested to build the tool, using the prompt:

```
1.Search in x / twitter about the terms "eth", "bitcoin", "solana" and
"cardano".
2.Using an LLM analyze the messages generated and categorize them.
3.Then using an LLM do a summary per category.
```



The synthesize of this code from the prompt, using Shinkai's planner took only two minutes. Now, the tool can be given metadata and connected to the user's personal tool database for later use:

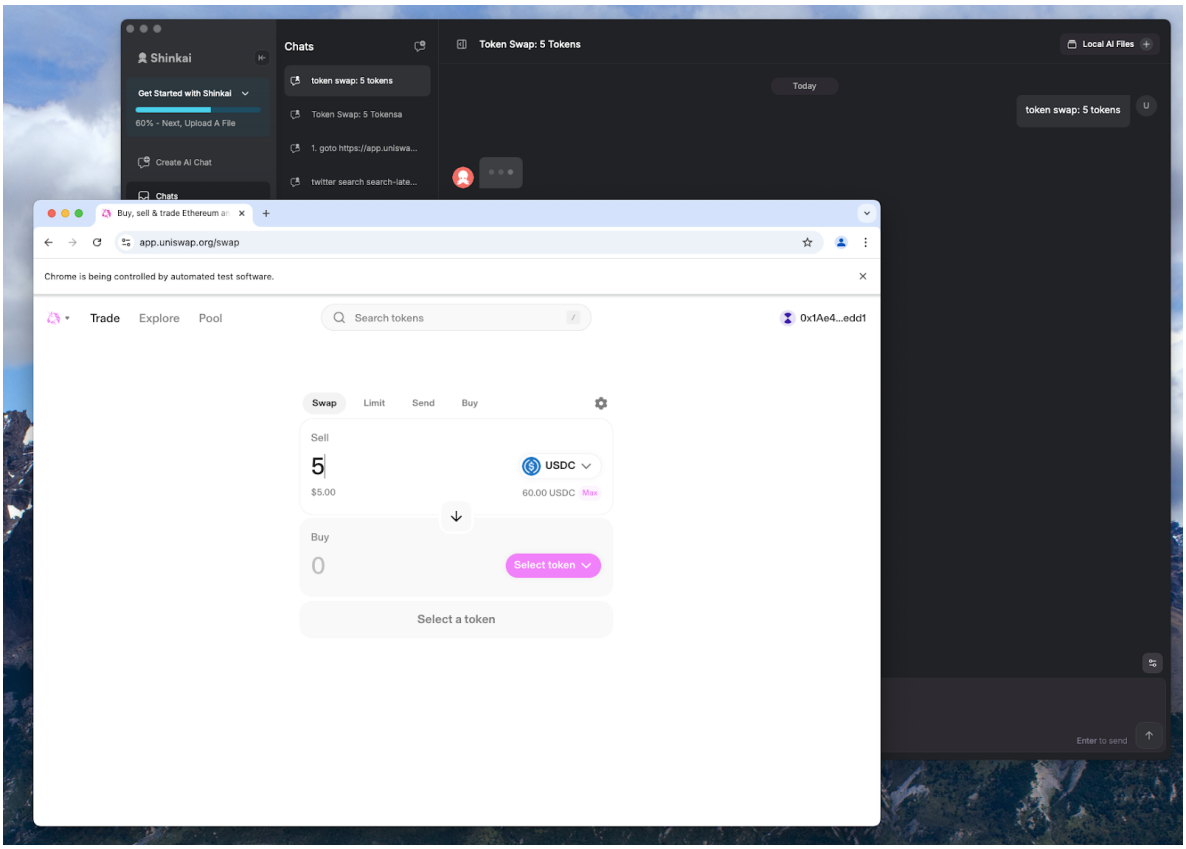


Optionally, the tool can be shared and this metadata will enable other users to make use of it.

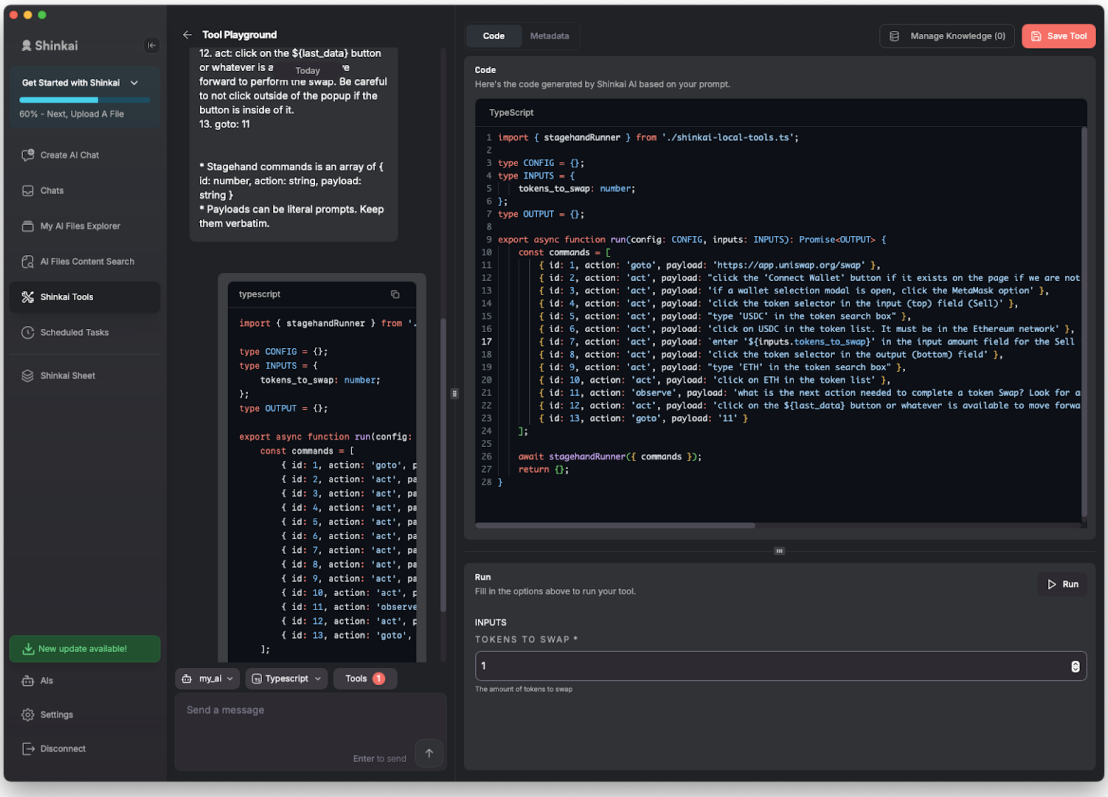
Automated Uniswap Interface

For our final tool, we show an agent-enabled interface to Uniswap. In this case, the user can request the agent to perform certain Uniswap actions, which it does by controlling another browser window and interacting with a crypto wallet and DApp. Like the other examples, this tool is reusable, so that a user could, for example, connect the two tools together and say something like "Buy $1,000 of whatever crypto token is trending most on Twitter today".

The first screenshot shows the Shinkai interaction (in the background; it reads "token swap 5 tokens") and the controlled window (in the foreground):



This tool is created by the user requesting Shinkai to make a tool that orchestrates the browser using the Stagehand library:



This is a more involved example where the user has to understand a bit more about the "programming" aspect of the tool, because in our tool creation prompt we gave a detailed, written description of the Stagehand commands. This is the sort of the tool that is more likely to be released on the network and imported by casual users.