Markdown Agent OS (MAOS): Building Multi-Agent Systems from Markdown, Terminals, and Coding Agents

Mohammed Reschreiter

Abstract

Coding agents are usually presented as tools for writing or refactoring code in IDEs. Multi-agent frameworks extend this idea but often remain locked behind programming APIs, configuration files, and infrastructure. This paper explores an alternative: building complex agent workflows using only plain-language Markdown files stored on a user's computer. We refer to this pattern as **Markdown Agent OS (MAOS)**.

In MAOS, the workflow is the document: tasks, steps, constraints, and expected outputs are written as Markdown. An agent runtime with local tool access (file I/O and command execution) interprets these documents and carries out the work in the same workspace. This shifts orchestration from code into editable text, turning ordinary folders into inspectable, portable "agent programs".

We argue that this simple idea—*Markdown as the control layer for local agents*—has practical advantages: it lowers the barrier to authoring automations, makes workflows easy to audit and version, and keeps long-lived state in human-readable files rather than hidden system prompts or opaque backends.

1 Introduction

Large language models have made "coding agents" a normal part of daily development. Tools like Claude Code and similar CLIs can read and write files, run shell commands, and help developers refactor or generate code on demand. At the same time, multi-agent frameworks promise more complex behavior: specialized agents that collaborate, plan, call tools, and maintain long-running workflows.

Despite this progress, the way we orchestrate agents is still mostly code. Defining workflows usually means writing Python graphs, configuration files, pipelines, or custom backends. Even when a framework is powerful, the barrier to entry remains high: you must understand its abstractions, manage dependencies, deploy services, and debug both your code and the model's behaviour. For non-technical users this is effectively impossible; for technical users it is slow and fragile.

This work starts from a different question:

- What if we do **not** build a new framework at all?

- What if we take the tools we already have—a terminal, a filesystem, and a coding agent CLI with access to tools and MCP servers—and treat them as the runtime for a purely text-defined system?

We call this approach **Markdown Agent OS (MAOS)**.

The core idea is simple:

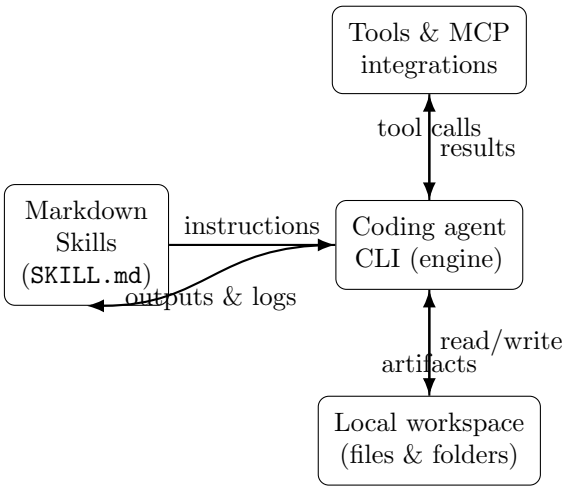

Figure 1: MAOS at a glance: Markdown skills describe workflows; a coding agent CLI executes them using local tools and files.

- The terminal + filesystem are the **body**.
- The coding agent CLI is the **brain**.
- Markdown files are the **nervous system**: they define skills, workflows, and constraints in plain language.
- Tools and MCP integrations are the **hands**: they allow the agent to browse, query APIs, update databases, and call external CLIs.

In MAOS, the word *skill* has two layers:

- Conceptually, a skill is a natural-language description of a repeatable workflow.
- Concretely, in this implementation, skills are realised as **Claude Agent Skills**: folders that contain a `SKILL.md` file plus optional reference documents, scripts, and other resources. Claude Code can load and use these skills natively as part of its normal operation.

A *flow* in this system is a short trigger (for example `/new_idea` or `/client_research`) that tells the agent which skill to execute and with which inputs. Once the skill is written, no further coding

is required: the agent reads the instructions, uses its existing capabilities over the filesystem and tools, and carries out the workflow.

In practice, this turns the terminal into a kind of natural-language operating system. Workflows for writing books, analysing markets, auditing client websites, or publishing blog posts to a database are all expressed in the same medium: plain text. Moving the system to another machine means copying a folder. Updating the logic means editing Markdown.

This paper describes that system, the kinds of workflows it can express, and the early results from using it in real work: from writing a book draft to analysing a client's website and generating a redesign report in a single, repeatable run.

2 Contribution

This paper contributes a *pattern* for local-first agent systems: represent the logic of an automation as ordinary Markdown files that live alongside the project's inputs and outputs.

Specifically, the paper argues that:

- **Markdown can act as the orchestration layer.** A workflow can be specified as plain-language steps, constraints, and output contracts in a document that both humans and agents can read and edit.
- **The local workspace is the runtime.** Instead of introducing a new framework, the runtime can be a standard filesystem plus an agent that is allowed to read/write files and execute tools with user-granted permissions.
- **File-based workflows are portable and inspectable.** "State" becomes a set of human-readable artifacts (notes, drafts, logs, checklists) rather than hidden memory, enabling review, versioning, and reuse.
- **The approach is model- and tool-agnostic.** Any agent runtime capable of reading Markdown instructions and operating on local files can implement the pattern.

The goal is not to prescribe a specific directory structure, vendor, or integration stack, but to articulate why treating Markdown as the control surface for agentic work is a useful design idea.

3 Concept: Markdown-Defined, Skill-Driven Agent Systems

The core concept of this work is to treat a terminal, a filesystem, and one or more coding agent CLIs as a complete runtime for multi-agent systems—without introducing any additional framework or programming language. All orchestration logic lives in plain-language Markdown files, which we call *skills*. We refer to this overall pattern as **Markdown Agent OS (MAOS)**.

For clarity, we use the following terms:

- **Workspace** — a folder on disk implementing one MAOS system (for example, a client research workspace or blog workspace).
- **Skill** — in MAOS, a Markdown-defined workflow. In this implementation, skills are realised as Claude Agent Skills: directories containing `SKILL.md` and optional support files.
- **Flow** — a user-facing entry point such as a slash-command (e.g. `/client_research`, `/start-blog-research`) that triggers one or more skills.
- **Engine** — a coding agent CLI instance (for example Claude Code) or a sub-agent CLI (such as Codex, Gemini, or Qwen) used by a skill.

In this pattern, coding agents are responsible for interpreting skills, calling tools, and reading and writing files.

3.1 Terminal + Filesystem as the Runtime

Instead of running agents on a server or inside a custom backend, the system assumes a very simple environment:

- A terminal where the user can invoke commands or slash-commands.
- A project folder on the local filesystem.
- One or more coding agent CLIs that can:
  - read and write files,
  - execute shell commands,
  - call tools via MCP or similar protocols.

This environment is already familiar to developers and power-users. It requires no additional infrastructure: no databases, no web services, no deployment pipeline. The "state" of the system is just the set of files and folders in the project directory.

3.2 Skills and Agent "Skills"

In MAOS, skills are the primary orchestration mechanism. Conceptually, a skill is a natural-language description of a repeatable workflow: what to do, in which order, with which tools, and where to store the results.

In practice, we implement a skill as a *workflow-in-a-folder*: a directory whose name identifies the skill and which contains a `SKILL.md` file plus optional reference documents, templates, and helper scripts.

One concrete implementation is Anthropic's **Claude Agent Skills** system in Claude and Claude Code. In that system, an Agent Skill is:

- a directory whose name identifies the skill,
- containing at minimum a `SKILL.md` file with:
  - a YAML front matter block (name, description, optional metadata such as version and dependencies),
  - and a Markdown body with detailed instructions, examples, and guidance.

The skill directory may also include additional Markdown files (for example `REFERENCE.md` or style guides), helper scripts written in languages such as Python or shell, and templates or other resources that the agent can read or execute when the skill is active.

Although skills originated as a vendor-specific feature—introduced by Anthropic in the Claude product ecosystem [1]—the folder-based pattern has begun to generalise. In our own practice, essentially the same `SKILL.md`-centred "workflow-in-a-folder" convention can be executed with multiple agent runtimes (e.g. Codex, Claude Code, Gemini CLI, Qwen Code CLI, OpenCode, Cline, Kilo, and others), because the core contract is simply: (1) load instructions from files, (2) operate on a shared workspace, and (3) write outputs back to disk.

In MAOS, this matters because it turns skills into a *portable interface*: when the workflow is expressed as readable files (rather than code), teams can reuse the same logic across different agent runtimes with minimal translation.

3.3 LLM-Agnostic and Multi-LLM Collaboration

MAOS allows for **multi-LLM collaboration**, where different language models—such as Claude, Codex, Gemini, or Qwen—participate within a single coordinated workflow. In this setup, **Claude Code** typically acts as the primary orchestrator because it supports Agent Skills, can read and write files directly, and provides deep tool integration. Other CLIs are treated as specialised sub-agents that execute specific tasks, often in parallel.

Each sub-agent is invoked from within a skill using a shell or MCP command. These sub-agents operate **headlessly**, meaning they run autonomously without user interaction, and they handle distinct subtasks like research, data extraction, or analysis. Their outputs are always written to structured file locations inside the shared project workspace—Markdown, JSON, or CSV files depending on the task.

Once the sub-agents complete their assignments, the main agent (Claude Code) reads the generated files, cross-compares the results, and synthesises them into a single, cohesive deliverable. For example, when running a `/start-blog-research` flow, Claude might:

1. Trigger Codex to perform a technical literature scan.
2. Ask Gemini to summarise mainstream sources and related trends.
3. Instruct Qwen to find recent community or open-source discussions.

After all three finish, Claude aggregates the findings, filters redundancy, evaluates conflicting claims, and writes a unified research summary or first draft. The user can then review or edit the result before publication.

This design achieves a **multi-LLM orchestration layer without any explicit orchestration code**. There are no agent graphs, sockets, or external schedulers—coordination emerges naturally through the shared filesystem and plain-text instructions. Markdown skills define how agents collaborate, what each should produce, and where outputs are stored. The filesystem itself acts as both shared memory and message bus.

Because the pattern is **LLM-agnostic**, adding or replacing models is straightforward. Any CLI-capable model that can read from and write to files can participate. The same Markdown-based workflow can run with different model combinations, enabling flexible experimentation and hybrid reasoning setups where each model contributes its unique strengths.

In essence, MAOS treats the filesystem as the lingua franca between agents, the Markdown skill as the conductor's score, and the coding agent CLI as the orchestra leader.

3.4 A Minimal Formal Model

This section provides a compact, slightly formal description of MAOS. The goal is not heavy mathematics; it is simply a precise way to say "the files are the state, the skill folder is the program, and the agent runtime executes it."

Artifacts.

- **Workspace** W: a directory containing all inputs, intermediate results, and outputs.
- **Skills** $S = \{s_1, \ldots, s_n\}$: each skill s_i is a directory containing at least `SKILL.md` (instructions) and optional helpers (templates, reference docs, scripts).
- **Engine** E: an agent runtime that can read/write files and execute tools/commands with the permissions granted by the user.
- **State** σ: the current set of files in W (including logs), plus any external systems reachable via tools (e.g. a database).

Execution cycle. A flow invocation selects a skill $s \in S$ and an input set I (often a URL, topic, or folder). The engine E then:

1. reads `SKILL.md` and relevant workspace files,
2. performs a sequence of tool calls and file writes,
3. terminates by writing outputs into agreed locations in W.

Gating and commit semantics. For workflows that write to external systems (e.g. databases), MAOS commonly uses an explicit *approval gate*: the engine first writes a human-readable draft (e.g. `content.md`); only after explicit approval does it emit "commit" artifacts (e.g. SQL files) or perform irreversible tool actions.

Memory. "Memory" is treated as ordinary workspace state: durable Markdown files that record preferences, project constraints, and prior decisions. This makes the system inspectable (humans can read it) and debuggable (agents can diff it).

3.5 From Chat Interface to Natural-Language OS

In this setup, the coding agent CLI is no longer just a chat interface. It behaves more like a shell that understands natural language, where:

- flows are short commands (e.g. `/new_idea`),
- programs are skills implemented as folders with `SKILL.md` and supporting files,
- tools and MCP connections act like system calls,
- and the filesystem is the shared memory between agents.

The result is a kind of **natural-language OS** running entirely inside the terminal. Complex automations—such as writing a book from idea to final draft, performing a full client website audit, or researching and publishing blog posts directly into a database—are expressed in the same medium that humans already use to think and communicate: written language.

4 Design Principles and Illustrative Vignettes

MAOS is intentionally described at the level of an idea rather than an implementation. The central claim is that a local agent can be made more useful and more controllable when its workflows are expressed as editable Markdown artifacts that live next to the work itself.

4.1 Design Principles

The pattern is guided by a small set of principles:

- **Text is the control surface.** The workflow is a document that can be read, reviewed, and edited like any other writing.
- **Files are the state.** Inputs, intermediate notes, drafts, and logs are stored as ordinary artifacts rather than hidden "memory".
- **Small contracts beat big frameworks.** A workflow specifies what the agent should produce and where it should write it; the agent runtime supplies the execution.
- **Portability via copying.** Reusing a workflow should be as simple as copying a folder of Markdown files into a new workspace.
- **Human gates are first-class.** When an action is irreversible or high-impact, the workflow should include an explicit approval checkpoint.

4.2 Illustrative Vignettes

The following vignettes illustrate the idea without prescribing a particular folder layout or tool stack.

Vignette 1: Writing pipeline. A user maintains a Markdown workflow that describes a repeatable process for turning a topic into an outline, a draft, a revision checklist, and a final publish-ready document. The agent reads the workflow, creates the specified artifacts, and stops for approval at defined checkpoints.

Vignette 2: Research workspace. A user keeps a "research" workflow in Markdown that instructs an agent to gather sources, store excerpts and notes in a structured way, and produce a short synthesis with citations. Because the notes and synthesis are files, the user can audit what was used and reuse the workspace later.

Vignette 3: Local operations. A user encodes a lightweight operational procedure in Markdown (e.g. "rename a batch of files, update references, and produce a diff report"). The agent executes it locally and writes a summary report, making the run inspectable and repeatable.

4.3 Limitations and Safety Considerations

MAOS inherits both the strengths and weaknesses of natural-language execution. Agents may interpret instructions differently than intended, and tool calls may fail or change over time. Additionally, when workflows can instruct an agent to execute commands and edit files, third-party workflows must be treated as untrusted code.

A practical MAOS workflow therefore often includes:

- explicit scopes (which folders may be read/written),
- checkpoints before destructive steps,
- and lightweight logs that record what the agent changed.

5 Future Work

Several directions emerged naturally while using the system and observing its strengths and weaknesses.

5.1 Project Templates and Init Skills

A recurring pattern is the need to create similar structures across projects: `.claude/skills`, provider-specific folders, temporary content directories, and optional archives or logs. A natural extension is a desktop application or CLI that scaffolds new projects with a standard layout, generates an initial skill that interviews the user about the project goals, and automatically creates flows and helper skills based on those answers.

This would make the system approachable for non-technical users and reduce setup friction even further.

5.2 Systematic Logging and Feedback Loops

Another direction is more systematic logging and feedback collection. Standardising a `logs/` directory and a feedback file per workflow would allow automatic recording of key steps and decisions, structured user feedback after each run, and skills that read past feedback to adjust behaviour over time.

This would formalise the "learning" that already happens informally when the user edits skills after each run.

5.3 Cost and Speed Optimisation

As multi-LLM workflows grow, cost and latency become important. Future work includes shifting more research tasks to cheaper or open-source models where possible, using the primary agent mainly for coordination and synthesis, and experimenting with smaller models as they become more capable.

5.4 Optional UI Layer

Although the system is designed to live entirely in the terminal, many users are uncomfortable with command-line tools. An optional graphical interface could present projects, skills, and logs in a friendlier way, allow non-technical users to trigger workflows and review outputs, and still keep Markdown as the single source of truth underneath.

Such a UI would not replace the terminal-based approach but sit on top of it, preserving the portability and transparency of the underlying text-based design.

6 Related Work

This work sits at the intersection of several existing tool categories but differs from each in important ways.

Multi-agent frameworks (e.g. LangGraph, AutoGen). Multi-agent frameworks provide programmable abstractions for defining agent graphs, message routing, memory, and tool usage [2, 4]. They are powerful but require writing and maintaining code, managing dependencies, and often deploying infrastructure. By contrast, the approach described here uses no framework code at all: orchestration lives entirely in skills, and the runtime is a coding agent CLI plus the local filesystem. It trades fine-grained programmatic control for accessibility, portability, and ease of iteration.

IDE-integrated copilots and coding assistants. Tools such as IDE copilots focus on assisting with code inside an editor: completing functions, suggesting refactors, or generating boilerplate. Some can run shell commands or edit multiple files, but their primary mental model is "help me write code." In this work, coding agents are repurposed as general operators over the filesystem and tools, with no requirement that they ever emit code. Skills describe research, analysis, and writing tasks; the agent behaves more like a research analyst or project collaborator than a code assistant.

No-code automation platforms (e.g. Zapier, Make). No-code tools allow users to connect services and define automation flows through graphical interfaces [5, 3]. They excel at integrating APIs but are limited by predefined blocks and often require careful UI work to maintain. The Markdown-based approach instead treats natural language as the workflow language. There is no visual editor; flows are written as text, executed locally, and can involve open-ended reasoning and content generation that go beyond typical trigger–action recipes.

Personal knowledge management tools with plugins (e.g. note-taking apps plus AI extensions). Some note-taking systems allow users to embed AI within their knowledge bases, generating or transforming notes. These tools bring models closer to the user's documents but usually keep automation inside the app, with limited access to arbitrary CLIs or MCP-style tools. The system described here inverts that relationship: the filesystem itself is the knowledge base, and the coding agent CLI orchestrates tools directly. Markdown is not just where knowledge is stored; it is also where process logic is defined.

Overall, the contribution is not a new framework or editor but a **pattern**: treating text files, coding agent CLIs, and tools as sufficient building blocks for multi-agent systems. This pattern can coexist with the tools above and, in some cases, wrap or drive them as part of a broader workflow.

7 Conclusion

This work introduced **Markdown Agent OS (MAOS)** and asked whether complex multi-agent workflows can be built without any orchestration code, using only plain-language Markdown files executed by coding agent CLIs with access to tools and MCP integrations. The experiments described here—client website research, proposal drafting, and multi-LLM blog creation—suggest that the answer is yes.

By treating the terminal and filesystem as the body, the coding agent CLI as the brain, and Markdown skills as the nervous system, it becomes possible to design powerful automations that remain transparent, portable, and easy to change. Workspaces live in a form that both humans and models can read and edit. Moving a system to another machine is copying a folder; improving it is editing text.

In practice, this pattern has reduced the time required for tasks such as client analysis and long-form content creation from hours or days to minutes or under an hour, while keeping a human in the loop for review and final decisions. The approach remains limited by model context and token constraints, occasional deviations from written plans, and the fragility of some tool integrations. Yet these limitations are manageable and likely to shrink as models and tooling improve.

Most importantly, Markdown Agent OS is not a product but a **way of working**: a philosophy of building multi-agent systems as collections of readable documents, rather than as opaque backends or complex graphs. This makes MAOS particularly attractive for non-technical users and teams who want to automate complex work without writing code. Workflows stay vendor-flexible and data stays on the user's own machines, so even if a specific model or provider disappears, the projects and documents remain usable.

Publishing this pattern and its concrete examples as an open repository invites others to adapt, extend, and challenge it—whether by creating shared skill packs, adding GUI layers, or integrating new types of tools. If multi-agent systems are to become everyday collaborators rather than specialised research projects, making them editable with the same tools we use for thinking and writing may be one of the simplest paths forward.

References

[1] Anthropic. Claude skills. Website, 2026. Accessed: 2026-02-08.

[2] LangChain, Inc. Langgraph (github repository). GitHub repository, 2026. Accessed: 2026-02-08.

[3] Make. Automation tool & integration platform. Website, 2026. Accessed: 2026-02-08.

[4] Microsoft. Autogen: A programming framework for agentic ai (github repository). GitHub repository, 2025. Accessed: 2026-02-08.

[5] Zapier. Zapier: Automate ai workflows, agents, and apps. Website, 2026. Accessed: 2026-02-08.