

Flash Coding: A Verified Agentic Coding Dataset Specification

Version 1.0*

Logitly AI
research@logitly.ai

Abstract

Agentic coding is a long-horizon problem. A single task may require dozens of tool calls interleaved with reasonings, and end in an outcome as small as a one-line edit or as large as a rewrite across many files. Training models on this behavior requires traces of the full course of the work: agentic data that can be trusted. We present Flash Coding, a dataset of agentic coding traces comprising two parts that share a single record schema. The supervised fine-tuning (SFT) part contains one trace per task, a successful completion; the direct preference optimization (DPO) part is contrastive, pairing a chosen trace that completes the task with a rejected one that does not. Every trace is produced under a controlled runtime. An automated verification protocol then validates each record, making the data human-aligned and machine-verified. This specification defines the record and trace format, the intended usage, the verification protocol, and the recommended evaluation.

Keywords

Agentic Coding, Terminal Use, Dataset Specification

1 Overview

A coding agent operates in an iterative loop that begins with a user query, which may be underspecified. The agent explores a given code repository, inspects the code to identify the root cause, applies changes through file edits, and invokes build and test commands to verify the result. This execution is long-horizon: a single task can require dozens of tool calls interleaved with reasoning, and can terminate in an outcome ranging from a one-line edit to a rewrite spanning many files. A failure at any step can invalidate the entire task.

Training a model to execute this loop requires traces of the loop itself. We present Flash Coding, a dataset that provides such traces: records of a coding agent working in real source repositories. Every trace is produced under a controlled runtime: a sandboxed environment, a pinned repository, a fixed working directory, and a restricted toolset and mode, so every action in the trace is attributable and checkable. An automated verification protocol then validates every record (§4), making the data human-aligned and machine-verified.

Flash Coding comprises two parts that share a single record schema. The supervised fine-tuning (SFT) part provides one demonstration trace per task; the direct preference optimization (DPO) part provides a chosen and a rejected trace of the same task. Table 1 gives a glance at one record: its context fields, its task, and the summary the agent writes for the user. The tasks are simulated user queries in multiple languages: the task text and the agent’s replies follow the task’s language (§6).

Table 1: Example record.

id	linux-1922043826
mode	plan
toolset	[read, write, edit, bash]
workdir	/home/develop/linux
init_md	<An AGENTS.md describing the Linux code structure>
task	USB fails to resume properly on Raspberry Pi and other Broadcom SoCs after suspend. The system log shows “nobody cared” IRQ errors, and the USB interface is left unusable.
summary	<The agent finds the root cause in a misconfigured dwc2_set_bcm_params and writes a plan to fix the bug>

Table 2: Record fields.

Field	Definition
id	A repository name plus a number. The same ID means the same task in every part.
mode	The agent’s operating mode is one of work, plan, or cplan. Here cplan is a file-backed plan mode: the agent writes the plan to a file.
toolset	The names of the tools the agent may call.
workdir	The agent’s working directory, as an absolute path.
init_md	A Markdown document about the repository: overview, directory layout, coding conventions, build commands, version. It may be empty.
messages	SFT part only. One demonstration trace for the task.
chosen	DPO part only. The preferred trace of the task.
rejected	DPO part only. The dispreferred trace of the same task.

2 Record and Trace Format

Records are JSON Lines: UTF-8, one record per line. Table 2 defines the fields of one record. The first five fields describe the environment and are shared by both parts; the rest carry the traces. We call the content of messages, chosen, or rejected a *trace*, the environment fields the *context*, and the chosen and rejected traces of one DPO record a *pair*.

A trace is a list of messages in the style of the Anthropic Messages API: each message has a role (user or assistant) and a content, a list of typed blocks. The messages strictly alternate between the two roles, starting with user and ending with assistant. Listing 1 shows an abridged trace.

The four block types:

- **text**: the task in the first user message; the assistant’s narration between tool calls, or the final summary.

*This document is a dataset specification, not a research paper; do not distribute.

Listing 1: An abridged trace.

```
[
  { "role": "user",
    "content": [ { "type": "text",
                  "text": "<task text>" } ] },
  { "role": "assistant",
    "content": [
      { "type": "thinking", "thinking": "<reasoning>" },
      { "type": "tool_use",
        "id": "tool_use_1", "name": "glob",
        "input": { "pattern": "**/*active*job*" } },
      { "type": "tool_use",
        "id": "tool_use_2", "name": "grep",
        "input": { "include": "*.md",
                  "pattern": "Solid Queue" } } ] },
  { "role": "user",
    "content": [
      { "type": "tool_result",
        "tool_use_id": "tool_use_1",
        "is_error": false,
        "content": [ { "type": "text",
                      "text": "(no match)" } ] },
      { "type": "tool_result",
        "tool_use_id": "tool_use_2",
        "is_error": false,
        "content": [ { "type": "text",
                      "text": "<matching lines>" } ] } ] },
  ...
  { "role": "assistant",
    "content": [
      { "type": "thinking", "thinking": "<reasoning>" },
      { "type": "text", "text": "<summary text>" } ] }
]
```

- **thinking**: the agent’s private reasoning for the turn.
- **tool_use**: a request to run one tool from toolset; input is a JSON object.
- **tool_result**: the outcome of one tool_use, matched by id; is_error marks a failed call.

We use the Anthropic message format because it is well structured. The traces are not distilled from Anthropic models: they carry no signature and no redacted_thinking blocks, and the thinking field is plain text.

The two sides have fixed shapes:

- **User side**. The first message opens with one text block: the task. Every later user message carries only tool_result blocks, one per tool_use of the preceding assistant message.
- **Assistant side**. Each message holds one thinking block, then any mix of tool_use and text blocks in generation order. The last message closes the trace with a text block: a summary written for the user.

3 Tool Definition

We define the toolset in Table 3 and its parameters in Table 4.

4 Verification Protocol

Verification is post-hoc: it runs on the finished record, after the controlled runtime has produced the trace, and a record enters the dataset only if it passes. The protocol works in two directions:

Meta checks. The record conforms to the schema and the trace format of §2: the messages strictly alternate between user and assistant, and the tool-call loop is closed: every tool_use carries

Table 3: The toolset.

Tool	Description
glob	Match paths against a glob pattern.
grep	Search file contents using regex.
read	Read contents from a file.
write	Write contents to a file, creating or overwriting it entirely.
edit	Replace a specific string in a file with new content.
bash	Execute a shell command.

Table 4: Tool parameters.

Tool	Parameter	Type	Description
glob	pattern	string	The glob pattern to match.
grep	pattern	string	The regex to search for.
	include	string	Optional file pattern filter.
read	path	string	Path of the file to read (absolute or relative).
	offset	integer	Optional line to start reading from, 1-indexed.
	limit	integer	Optional maximum lines read, 2000 by default.
write	path	string	Path of the file to write (absolute or relative).
	content	string	The content to write.
edit	path	string	Path of the file to edit.
	old_string	string	The unique string to find and replace.
	new_string	string	The replacement string.
	replace_all	boolean	Optional; replace every occurrence, false by default.
bash	command	string	The shell command to execute.
	timeout	integer	Optional timeout in milliseconds, 120000 by default.

an id unique within the trace and is answered by exactly one matching tool_result; a call for a tool outside toolset is refused with an error. In a DPO pair, the two traces differ. The task passes two further checks: it is solvable, so a trace’s outcome reflects the work rather than an impossible task; and it sets one language, which the agent’s replies follow without mixing.

Task completeness. Each trace is assessed for task completeness: whether the work finishes the task. The assessment is machine verification from end to end and is applied against a human-aligned standard: every trace in the dataset is judged by the same bar, which keeps the verdicts consistent and the data reproducible at scale.

Beyond single records, the protocol also enforces dataset-level properties: each id appears only once within a part, and an id that appears in both parts carries the same context in both. Together, the two directions make the data human-aligned and machine-verified.

5 Intended Usage

The two parts carry two signals, and each matches its objective. In the SFT part, the trace is a demonstration: behavior to imitate, with no score attached. In the DPO part, the label is an ordering: within a pair, chosen is preferred over rejected on the same task and context. This is the only claim the dataset makes about the pair:

chosen is not certified correct, and the reason for the preference is not in the data. DPO consumes exactly this ordering, and nothing more. No other labels exist (§8).

The two parts are not disjoint by task. An id identifies a task: when an SFT record and a DPO record share the same id, they are the same task, and the same trace may serve as messages and as chosen. Training on both parts therefore sees some tasks twice, under two objectives.

6 Known Characteristics

The following properties of the data are worth stating explicitly.

System prompt. By design, the dataset does not record a system prompt: a trace begins with the task itself, as the first user message (§2). The prompt behind a trace is nevertheless reconstructible from the context fields: it declares the role of a coding agent, gives the working directory `workdir`, appends `init_md`, and enables the tools of `toolset`, defined in §3. We recommend randomizing the reconstructed prompt at training time, so a model does not overfit to one fixed wording: give the agent a different name and a differently worded role statement, and declare the tools of `toolset` in a different order.

Thinking. By design, the thinking block must be present at every assistant message, including the messages that answer tool results. The traces therefore carry the complete chain of thought: the thinking field is plain text, not encrypted, not summarized, and not rewritten (§2).

Adaptive thinking. Recent models practice adaptive thinking: the model itself decides when reasoning helps and omits it otherwise. The dataset supports the same behavior without altering any record: at training time, drop a thinking block when it is short, or when the assistant’s reply says roughly what the thinking block already says. The same traces therefore serve models that always think and models that think only when needed.

Error flag. Every `tool_result` carries `is_error`, which marks a call that did not exit 0. A flagged result is not damaged data: a nonzero exit can be the ordinary outcome of a call. A `grep` run through a shell, for example, exits 1 when nothing matches, and such a call is reported with `is_error=true`. Keeping these results is deliberate: the model reads a failure and learns to recover from one over long-horizon work.

Language. Tasks are written in multiple languages. In a positive trace (messages, or chosen in a pair), the assistant replies in the task’s language; the language of thinking is not defined.

Cheating. Agentic models are known to cheat on repository tasks: instead of doing the work, an agent reads the solution out of the environment, for example by inspecting leaked git pack files that carry the upstream fix, or by downloading the patch from the project’s website. Both channels are carefully controlled and cannot happen in Flash Coding.

Plan mode. The dataset’s plan mode is stricter than the common convention. In plan mode, writing files is strictly forbidden, and `/tmp` is not writable either. The one exception is `/dev/null`, which stays writable, because writing to it involves no real file change.

Toolset. Each record has its own toolset, a combination of shell, file match, and read/write tools (§3). Within a record the toolset is

mode-invariant: plan mode sees the same tools as work mode, so a mode change alters no prompt prefix. The mode gates use, not existence: in plan mode, a call that modifies files cannot complete.

7 Recommended Evaluation

Flash Coding is training data (§5): it carries no scores and no task-success labels (§8), so the effect of training on it cannot be read off the data itself. We recommend measuring the effect on external benchmarks, and we recommend two: DeepSWE and Terminal-Bench. The dataset shares no tasks with either benchmark: zero overlap, zero contamination, so a gain on either measures real capability, not leaked tasks. We expect training on Flash Coding to improve scores on both, for two reasons that map onto them: the traces demonstrate long-horizon task solving in a repository, and they demonstrate complex tool use, including shell use.

DeepSWE. A software engineering benchmark that grades an agent on implementing real features and changes inside existing codebases. Its tasks are written from scratch rather than taken from public pull requests, and each task is judged by a hand-written functional verifier that runs the code: success is decided by how the code behaves, not by matching a reference patch. This is the first reason: a Flash Coding trace is long-horizon work of exactly this kind, dozens of tool calls of exploration, edit, and verification inside one repository (§1).

Terminal-Bench. A benchmark of tasks executed through a command-line interface: the agent works in a sandboxed environment and must reach its goal through the shell, and automated tests grade the final state of the environment, so success depends on the outcome rather than the exact steps. This is the second reason: the toolsets pair search and file tools with shell access (`bash`), and a trace exercises the combination, chaining tools and recovering from calls that fail (§6).

8 Out of Scope

Scoring and verification belong to an internal pipeline: their detailed methods, and the raw scores they produce, are not released by design.