Agentic Engineering

Engineering practices for agentic AI systems

- Configuring Custom Models: Vision & Parallel Tool Calls
- Loop Engineering

Configuring Custom Models: Vision & Parallel Tool Calls

Overview

This page documents configuration patterns for using custom models (e.g. Qwen, custom fine-tunes) with **LiteLLM** as a proxy and **OpenCode** as the agentic frontend.

When using non-standard models, both LiteLLM and OpenCode require explicit configuration to advertise and support advanced capabilities like vision (image input) and parallel tool calls. Without proper setup, these features silently fail.

Vision / Multimodal Support for Custom Models

Configuring image support for custom models in a llama.cpp + LiteLLM + opencode setup.

The Problem

LiteLLM does **not** infer vision support for arbitrary `custom_openai` models. OpenCode also performs its own preflight modality check and does **not** read LiteLLM's `supports_vision` metadata for custom providers. Both layers must be configured independently.

LiteLLM Side

- Add `supports_vision: true` under the LiteLLM model's `model_info` so LiteLLM and its model metadata advertise image capability
- LiteLLM's OpenAI-compatible/custom OpenAI chat path does not normally strip OpenAI-style image blocks — it preserves `image_url` content and normalizes string image URLs into `{ "url": ... }`

OpenCode Side

- For opencode custom providers, image support must be declared per model with `modalities.input` including `image`
- If `modalities` is set, include both `input` and `output`
- Without `modalities.input: ["text", "image"]`, attachments get replaced with an error: `ERROR: Cannot read image (this model does not support image input). Inform the user.`

Working OpenCode Model Shape

```
"Qwen3.6-35B-A3B": {
  "name": "Qwen3.6-35B-A3B",
  "limit": {
    "context": 262144,
    "output": 65536
  },
  "modalities": {
    "input": ["text", "image"],
    "output": ["text"]
  }
}
```

Parallel Tool Calls

Enabling parallel tool calls for a custom model served through LiteLLM, used by OpenCode.

The Problem

The `parallel_tool_calls: true` in `model_info` is **metadata only** — it does not auto-forward to the API. Without explicit pass-through, the downstream server never receives the signal.

OpenCode Side

- **No config change needed** in `opencode.jsonc`
- OpenCode uses the AI SDK which auto-detects parallel tool call capability from the provider

- The `tool_call` boolean in model config is for declaring tool support, not parallel behavior
- Your setup (`jamesravey_litellm` provider with `@ai-sdk/openai-compatible`) handles this at the SDK level

LiteLLM Side

Fix: Add `parallel_tool_calls` inside `litellm_params`:

```
"litellm_params": {
  ...
  "parallel_tool_calls": true
}
```

Prerequisites

1. LiteLLM must be **v1.61.0+** (parallel tool calls pass-through added then)
2. The downstream inference server (vLLM, TGI, etc.) must support parallel tool calls

---

Files Reference

- `~/.config/opencode/opencode.jsonc` — OpenCode model config (no changes needed for parallel tool calls)
- LiteLLM model config — add `parallel_tool_calls` to `litellm_params` and `supports_vision: true` to `model_info`

Loop Engineering

Loop Engineering is a proposed approach to building software whereby an AI agent is used in a validation/feedback loop to essentially validate whether the task is complete yet or not.

The term is believed to have been coined by Addy Osmani, Peter Steinberger and Boris Cherny in June 2026. It is believed that this way of working emerged once agentic AI coding tools crossed a threshold:

> Coding agents had become reliable enough to finish a non-trivial task unattended; scheduling primitives had appeared in the major harnesses; and the cost of a single agent run had dropped far enough that running one repeatedly, on a timer, stopped looking wasteful.

Separate Generator and Evaluator Agents

The IEEE working note suggests that it is very important to separate the generator from the evaluator in the loop. The agents that generate code are typically very keen to praise themselves even when a human can plainly see that quality is mediocre. The context for the coding agent is stuffed full of reasons why the code was written in a particular way. So the agent looks at its own reasoning and sees a chain of self persuasion.

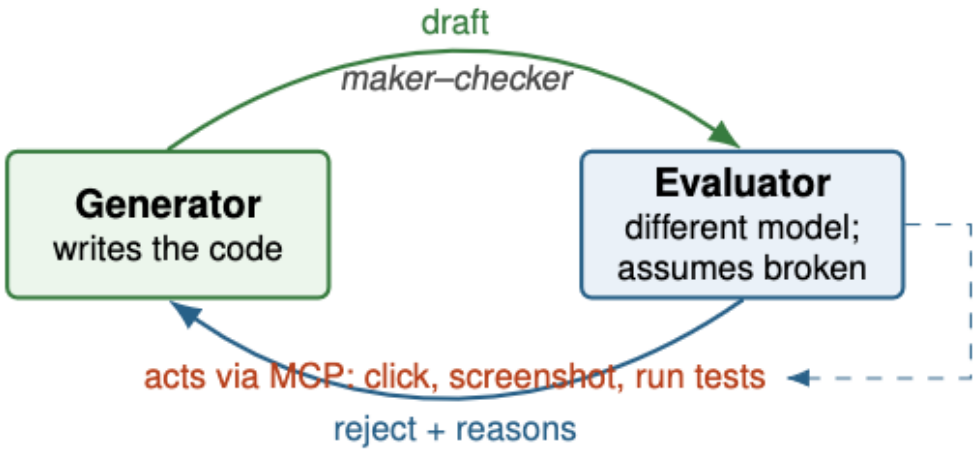


Fig. 3. Generator and evaluator as separate agents. The evaluator carries none of the generator's self-persuasion, defaults to doubt, and judges behavior by acting rather than just reading.

The key is to build an individual independent skeptic rather than trying to train the generator to be self critical. This is a similar architecture to the GAN model.

The evaluator should act, not just read. This implies that the model should do something like run tests, ideally end-to-end tests, that exercise the complete product. But they will also need to give enough feedback and granularity that the agent would know what to do with them afterwards.

References

- Addy Osmani Loop Engineering
- Loop-Engineering-IEEE.pdf