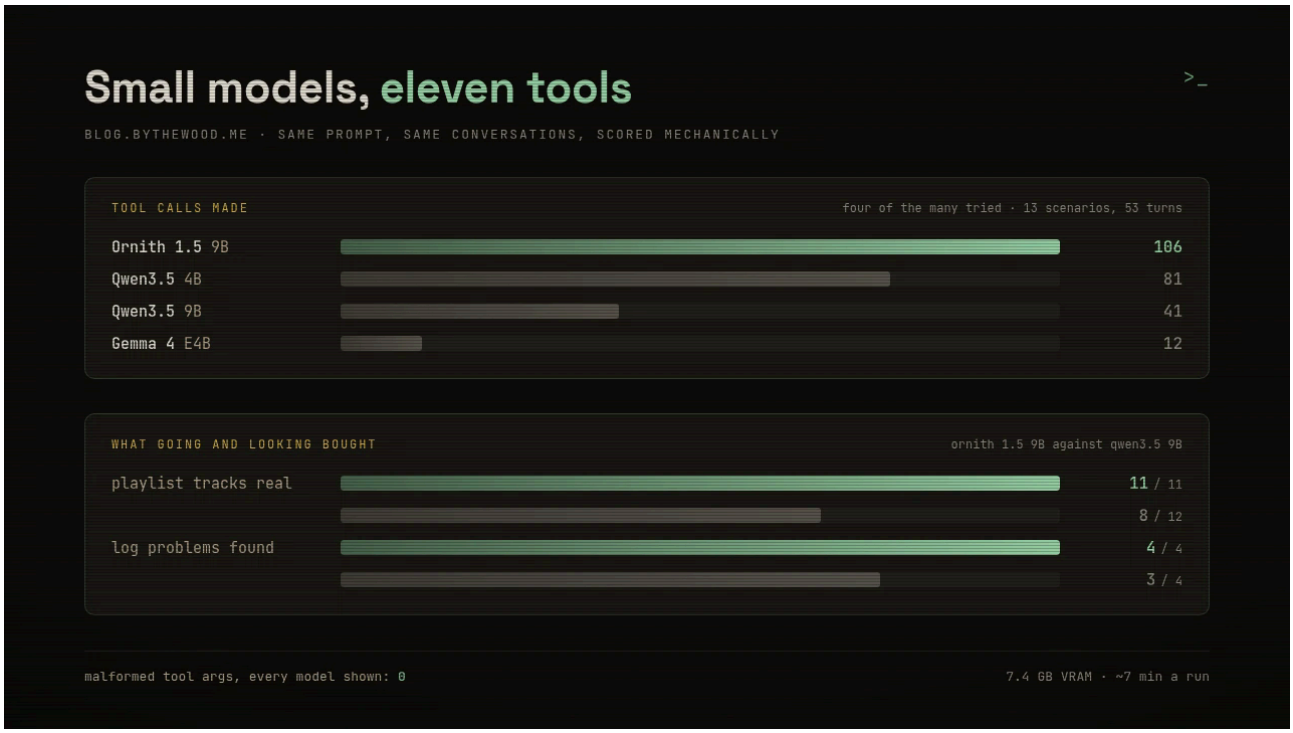
An agentic loop in Go for a small local model

Isaac Bythewood · 2026-09-20 · 9 min read

go ai webdev



I run my own general purpose assistant on a 9B at home, and the tool calling loop in it is about two hundred lines, most of which is stopping it telling me it could look something up if I wanted.

I run my own assistant at https://chat.bythewood.me/, the general purpose kind like Claude or ChatGPT, except it's a 9B on my own 3070 and everything it touches is mine. It has tools, attachments, history and a memory, and I use it for the same sorts of things I'd use one of the hosted ones for.

The agentic loop underneath is maybe two hundred lines of Go with no framework, and honestly the loop was the easy part. Nearly all of the work went into the thing that sits after it and decides whether the model actually answered.

The loop itself is what you'd expect. Offer the tool schemas, take what comes back, run any tool calls, append the results, go around again.

```
for round := 0; round < maxToolRounds; round++ {
  reply, st, err := e.llm.CompleteStats(ctx, msgs, offer, toolTurnTokens)
  if err != nil {
    return Message{}, used, stats, err
  }
  if len(reply.ToolCalls) == 0 {
    break // it wants to answer
  }
  msgs = append(msgs, reply)
  for _, tc := range reply.ToolCalls {
    args := json.RawMessage(tc.Function.Arguments)
    res := e.reg.Call(ctx, deps, tc.Function.Name, args)
    used = append(used, res)
    msgs = append(msgs, Message{Role: RoleTool, ToolCallID: tc.ID,
```

```
      Name: res.Name, Content: res.JSON()})
  }
}
```

That works fine with a big model. With a small one the `break` is where it all falls apart, because a reply with no tool call is not the same thing as an answer. What I kept getting was this:

> I don't have access to real time data, but I can search for the current price if you'd like.

The loop reads that as done. The user reads it as nothing. So the break goes through a gate first, and if the gate objects the turn goes back around with the tools still on the table.

```
if len(reply.ToolCalls) == 0 {
  if gates < maxGates && round < maxToolRounds-1 {
    nudge, gst := e.gate(ctx, user, reply.Content, answered, used, emit)
    if nudge != "" {
      gates++
      // The draft itself is never appended. A model handed
      // its own text back writes it again.
      msgs = append(msgs, Message{Role: RoleUser, Content: nudge})
      forceTools = true
      continue
    }
  }
  break
}
```

Two things in there matter more than they look. The draft never goes back into the conversation, since a model that can see its own deferral will write it again almost word for word. And `forceTools` flips the next round to `tool_choice: "required"`, which is what turns the nudge from a request into an instruction. Asking politely a second time gets you a politer deferral.

The gate runs its free checks before it spends a model call. Most deferrals are a handful of shapes and a regex catches them for nothing:

```
var deferrals = []*regexp.Regexp{
  regexp.MustCompile(`(?i)\b(let me|i'?ll|i'?m going to)\s+(go\s+)?(and\s+)?` +
    `(search|look|check|find|fetch)\b`),
  regexp.MustCompile(`(?i)\bwould you like me to\b`),
  regexp.MustCompile(`(?i)\bshall i\b`),
  regexp.MustCompile(`(?i)\bi (do not|don'?t) have ` +
    `(access to|real ?time|current|live)\b`),
  regexp.MustCompile(`(?i)\bmy (training data|knowledge) ` +
    `(only )?(goes|cuts off|ends)\b`),
}
```

Every pattern needs a first person subject or you throw away good answers, because "you can search for it on their site" is advice and not a deferral. I also only look at the first 240 characters, since an answer that does the work and then offers to check the other two things at the end has answered, and taking that away costs you the whole turn again.

When the cheap checks don't fire, the model gets asked. That call is constrained to a JSON schema, which llama.cpp compiles to a GBNF grammar and samples against, so an enum field can't come back as anything else:

```
type verdict struct {
  Verdict string `json:"verdict"` // "answered" or "research"
  Query   string `json:"query"`
}

var verdictSchema = map[string]any{
  "type": "object",
  "properties": map[string]any{
    "verdict": map[string]any{
      "type": "string", "enum": []string{"answered", "research"},
    },
    "query": map[string]any{"type": "string"},
  },
  "required":             []string{"verdict", "query"},
  "additionalProperties": false,
}
```

Two fields and one of them an enum. I tried giving it a free reasoning field next to the constrained one and it would write a paragraph arguing its way to one answer and then emit the other, which was maddening to watch. The query field is there because a verdict on its own isn't actionable, and a nudge that just says more research is needed sends the model back to the search it already ran.

The other thing that helped a lot is asking one question at a time. My first draft check weighed about eight rules at once and it was wrong often enough to be useless. Split into single judgements it got most of them right, and the freshness one is a whole prompt for one boolean:

```
Decide whether answering the user's question correctly needs information you
could not have from training alone, because it changes over time or has
happened since.

true when the question touches news, current events, prices, markets, scores,
odds, fixtures, schedules, opening or closing, weather, or what is happening now.
true whenever the question carries a time word like today, tonight, this
weekend, yesterday, right now, currently, or latest, even if the subject sounds
ordinary.

false when the answer is a definition, an explanation, how something works,
history, code, arithmetic, or a recipe, none of which change.
```

That one is asked of the question and never of the draft, because a draft that invented an answer reads exactly like one that knew it.

Two more things worth having if you try this. A model that gets a thin or failed result will ask for the same thing again, and again, until the budget is gone, so I keep a ledger of calls already made in this turn and answer a repeat from it with a line telling it not to:

```
key := tc.Function.Name + "\x00" + canonArgs(tc.Function.Arguments)
if prev, done := seen[key]; done {
  repeats++
  body, _ := json.Marshal(map[string]any{
    "repeat": true,
    "note": "You already called this tool with these exact arguments " +
      "in this turn. The result is below and it will not change. " +
      "Do not call it again. Use what you have, or try different " +
      "arguments, or answer and say what is missing.",
    "result": prev.Content,
  })
  msgs = append(msgs, Message{Role: RoleTool, ToolCallID: tc.ID,
```

```
    Name: prev.Name, Content: string(body)})
  continue
}
```

After two repeats I just stop offering tools at all, and on the last round I take them away and append a note saying what couldn't be found. Taking the tools off the table is the only reliable way I've found to make a small model stop fetching and write something, and it also means a tool that fails every time can't spin the turn out forever.

A note on the model, because picking the right one did more for this than any of the scaffolding above. I've run a lot of small models against this by now, all on the same system prompt, the same eleven tools and the same twenty odd conversations, scored mechanically rather than by asking another model what it thought. Four of them are worth putting side by side.

The one I landed on is Ornith 1.5 9B and I'd point anybody doing this at it. It made 106 tool calls across thirteen scenarios where Qwen3.5 9B made 41 and Gemma 4 E4B made 12. Not one of them ever emitted a malformed argument or invented a tool name, so the thing that tells them apart isn't whether they can format a call, it's whether they bother to go and look.

Three things it did that nothing else did:

- Asked for a playlist, it spent 65 calls checking every track against a music catalogue instead of trusting itself, and all eleven songs it gave me were real ones. The Qwen 9B trusted itself and invented four.
- Handed a log to read, it found all four of the problems I'd planted in it, including 64 errors on one endpoint that both Qwens walked straight past.
- With web search switched off it said so, then answered anyway with what it had and told me to treat the prices as ballpark. Gemma reads a missing tool as a full stop and refused eight of thirteen scenarios, which is most of why I'm not running it, since my tools fail fairly regularly.

Other people got here first

I built this off watching it fail rather than off any paper, and went looking afterwards to see whether anybody else had landed in the same place. They had, and some of it has been sitting in the literature for years.

- Self-RAG (Asai et al., 2023) trains a model to decide on demand whether it needs to go and retrieve anything at all, and then to critique its own draft for whether the evidence actually supports it. That is my freshness check and my gate, done properly as one trained model instead of two extra calls around a loop.
- CRITIC (Gou et al., 2023) takes a finished answer, has the model check it against a tool, and revises it from the feedback. Same shape as sending a draft back.
- Corrective RAG (Yan et al., 2024) puts a lightweight evaluator in front of the retrieved documents and kicks off a web search when they do not hold up. Their evaluator hands back something actionable for the same reason my gate returns a query.
- Reflexion (Shinn et al., 2023) writes feedback into the next attempt as plain text rather than touching the weights, which is what the nudge is.
- Efficient Guided Generation (Willard and Louf, 2023) is the theory under the enum trick, and it is the same idea as the GBNF grammars llama.cpp gives me for free.
- Small Language Models are the Future of Agentic AI (Belcak et al., 2025) makes the general case that a small model is the right size for most agent calls, which is the bet this whole thing is making.

The difference is mostly that those are training and framework answers and mine is a regex and a couple of constrained calls in the harness, since I am not fine tuning anything on a 3070. If you are building this properly, read those first.

The gate costs about a second on a turn that takes twenty, and it got the thing to stop offering to look stuff up, which is all I wanted out of it. The whole version is in orchard under `sites/chat.bythewood.me`, `chat.go` for the loop and `research.go` for the gate.