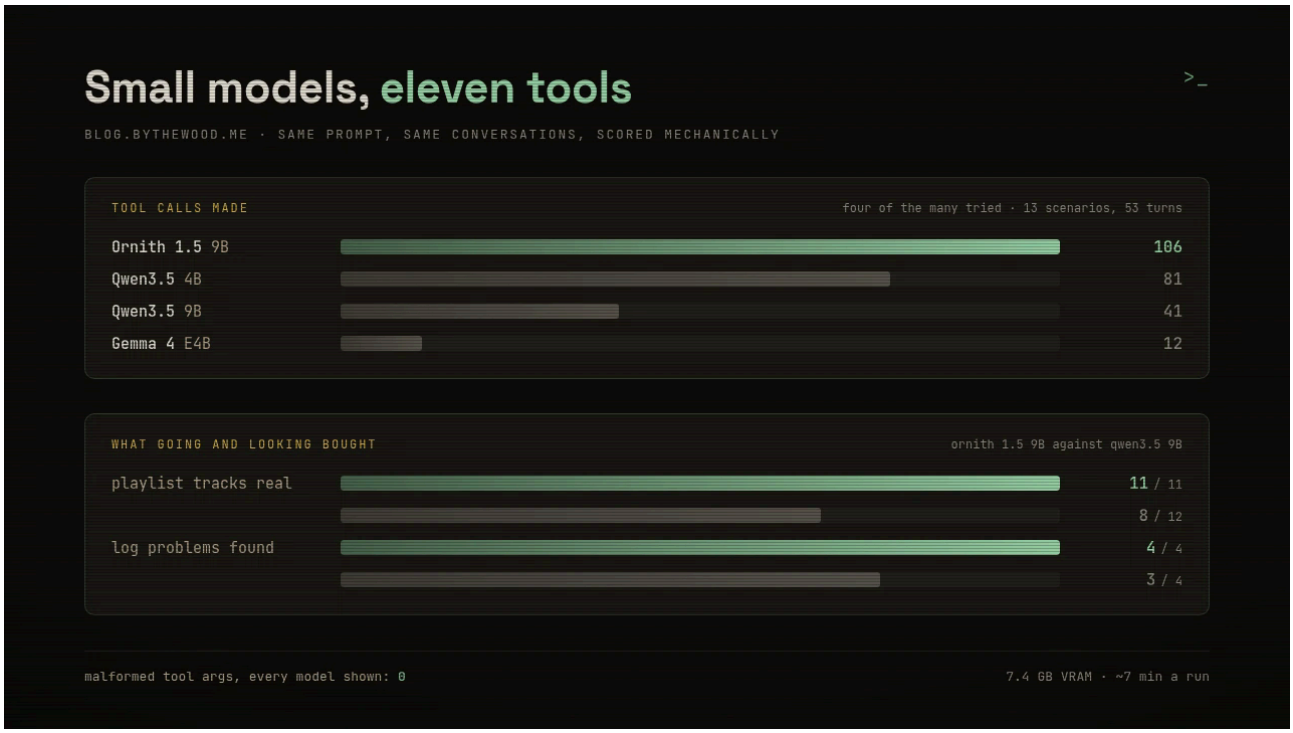
# An agentic loop in Go for a small local model

Isaac Bythewood · 2026-09-20 · 8 min read

go ai webdev




I run my own assistant on a 9B at home and the tool calling loop in it is a few hundred lines of Go, most of which is there to stop it offering to look something up instead of looking it up.

I run my own assistant at https://chat.bythewood.me/. It's the general purpose kind like Claude or ChatGPT except it's a 9B model running on my own 3070, with tools, attachments, history, and a memory. The agentic loop under it is a few hundred lines of Go with no framework and the loop itself was the easy part. Most of the work went into deciding whether the model actually answered when it stopped calling tools.

The basic loop is what you'd expect, offer the tool schemas, run any tool calls that come back, append the results, and 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's fine with a big model but with a small one the `break` is the problem, since a reply with no tool call isn't always an answer. I kept getting replies like this:

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

The loop treats that as done and I get nothing useful. So before the break the reply goes through a gate, and if the gate doesn't like it the turn goes back around with the tools still offered:

```
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
}
```

The draft never gets appended to the conversation because when a small model can see its own deferral it'll write it again almost word for word. `forceTools` sets `tool_choice: "required"` on the next round so the model has to call something, otherwise it tends to just defer again a bit more politely.

The gate tries a few free checks before it spends a model call. Most deferrals look about the same so a handful of regexes catch them:

```
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 start throwing away good answers, "you can search for it on their site" is advice and not the model putting off work. I also only check the first 240 characters, since an answer that does the work and then offers to check something else at the end did answer, and sending it back costs you the whole turn again.

When the regexes don't match the model gets asked whether the draft answered. That call is constrained to a JSON schema, which llama.cpp compiles into a GBNF grammar and samples against, so the enum 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,
}
```

I tried adding a free text reasoning field next to the verdict and it would write a paragraph arguing for one answer and then pick the other. The query field is there so the nudge can tell the model what to search for, if the nudge just says more research is needed the model runs the same search it already ran.

Asking one question at a time helped a lot too. My first version of the draft check weighed about eight rules at once and was wrong too often to be useful. Split into single questions it got most of them right. Here's the freshness prompt, which 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, this week, 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 only ever sees the question and never the draft, since a draft with a made up answer looks just as confident as one with a real answer.

Small models will also ask for the same thing over and over when a tool returns something thin or fails, until the round budget is gone. I keep a ledger of calls already made in the turn and answer a repeat from it with a note telling it to stop:

```
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 stop offering tools completely, and on the last round the tools come off and I append a note saying what couldn't be found. Taking the tools away is the only reliable way I've found to get a small model to stop fetching and write something, and it means a tool that fails every time can't spin a turn out forever.

## Picking the model

The model mattered more than any of this. I ran four small models against the same system prompt, the same eleven tools, and the same thirteen scenarios, scored with mechanical checks rather than having another model grade them. I went with Ornith 1.5 9B. It made 106 tool calls across the thirteen scenarios where Qwen3.5 9B made 41 and Gemma 4 E4B made 12. None of the four ever sent malformed arguments or made up a tool name, so what separated them was mostly how willing they were to go and check something.

Asked for a playlist, Ornith made 65 calls checking every track against a music catalogue and all eleven songs it gave me were real. Qwen 9B didn't check and made up four. Given a log with four problems planted in it Ornith found all four, including 64 errors on one endpoint that both Qwens missed. With web search turned off it said so and answered with what it had, telling me to treat the prices as ballpark. Gemma treated a missing tool as a reason to stop and refused eight of the thirteen, which rules it out for me since my tools fail fairly regularly.

## Related papers

I built this from watching it fail and only went looking afterwards, and a lot of it is already in the literature done properly with trained models instead of a regex and a couple of extra calls. Self-RAG is the closest, it trains a model to decide when to retrieve and to critique its own draft, which covers both my freshness check and the gate. CRITIC, Corrective RAG, and Reflexion are all variations on checking an answer and feeding text back into the next attempt. Efficient Guided Generation is the theory behind the constrained JSON, and Small Language Models are the Future of Agentic AI makes the case for small models in agents in general. If you're doing this seriously read those first.

The gate adds about a second to a turn that usually takes twenty and it got the model to stop offering to look things up. The code is in orchard under `sites/chat.bythewood.me`, `chat.go` for the loop and `research.go` for the gate.