

A goal-driven onboarding agent, and a scenario harness for tuning prompts

Working paper on Sprijin360 onboarding v2 and the prompt-audit skills for Claude Code

WORKING PAPER · EARLY RESULTS

Alin Rizea · DBM AI · alin.rizea@gmail.com · 16 October 2026

SUMMARY

This note describes two pieces of work. The first is the onboarding agent on Sprijin360, a directory of therapists in Romania. A practitioner talks to it for a few minutes and ends with a complete public profile. The agent is goal-driven: it keeps an explicit, scored list of what it still needs and aims every question at the weakest gap. It is built to run on a small, cheap, non-reasoning model (gpt-4o-mini), so the goal and the bookkeeping live in code and the model only does narrow language tasks. Each turn, one model call reads the reply into nine scored fields, plain code decides what is still missing, and a second model call writes the next question. At the end a LangGraph fan-out drafts every text in parallel. The second is prompt-audit: two Claude Code skills and a small Python harness that replay fixed scenarios through a real LLM pipeline, score each answer with an LLM judge, and track drift between runs. I describe how each one works, what testing found, and where the methods are still weak.

1. Context

Sprijin360 (sprijin360.ro) lists psychologists, psychotherapists, nutritionists, speech therapists and coaches. It grew out of LogoMio, a speech-therapy platform DBM AI built in 2025, and was built in two to three weeks in early 2026 on Django, LangGraph, PostgreSQL and Redis. About 40 therapists have registered.

The first onboarding (v1) was an eight-step wizard with a form per step. It worked, but it asked practitioners to write marketing copy about themselves, which most of them find slow and hard. Version 2 replaces the forms with a short conversation, mostly in Romanian, by text or voice. The system pulls structured data out of the answers and writes the profile for them. They review a pre-filled form next to a live preview, change what they want, and submit it for admin approval. v2 writes to the same models as v1 (Category, Service, Team, TeamMember) and reuses its save, review and publish code, so nothing downstream had to change.

2. The goal-driven onboarding agent

Design constraint: a cheap, non-reasoning model. The agent is built to run on a small, cheap model with no reasoning step: gpt-4o-mini, for all three roles (extractor, responder, writers). That keeps each onboarding cheap and each reply quick. A model in this class is good at narrow language tasks, such as pulling facts out of a paragraph into JSON or writing one question. It is unreliable at the things a reasoning model would manage on its own over a long chat: remembering what it already has, judging when it has enough, and noticing that it is asking the same thing again. So the model gets only narrow tasks, and the goal and the bookkeeping stay in code:

- One job per call, with a short, specific prompt. No call is asked to plan.
- The state is rebuilt and handed to the model every turn as compact JSON. The model is never trusted to remember it.
- The extractor and the writers return JSON, which is checked in code. A failed or malformed call falls back to an empty result or a stock reply instead of breaking the turn.
- Anything that can be counted is counted in code: service and schedule scores, turn limits, the stop rule, loop detection.
- Scores can only go up, so one bad extraction cannot undo earlier progress.
- Each writer sees only the fields it needs, so its context stays small and focused.

The goal itself (enough fields at 7 or more) is never handed to the model as something to pursue. It lives in the evaluator, and each turn the responder is only told which fields are still missing.

2.1 State: nine scored fields

The conversation fills a session record with nine fields, which the code calls *ingredients*. Each field holds the raw snippets the practitioner said, a clean merged version written by the model, and a score from 0 to 10. A field counts as covered at 7. Fields have a priority and one of three categories.

Table 1. The nine fields, in priority order.

#	Field	What it holds	Category
1	B_type	Type of practice: clinical psychology, psychotherapy, counselling, nutrition, physiotherapy, coaching	must
2	C_what_they_do	Services offered and the problems they help with	must
3	S_services_list	Structured list: name, price, duration, online or in person	must
4	D_audience	Typical clients: adults, children, couples, specific situations	should
5	E_usp	What sets them apart: approach, experience, hours, online sessions	should
6	H_specializations	Methods and specialisms: CBT, EMDR, systemic, trauma, addiction	should
7	S_schedule	Working days and hours	should
8	F_story	How the practice started, why they became a therapist	nice
9	G_approach	Therapeutic philosophy, what a typical session looks like	nice

2.2 One turn

Each turn makes two model calls with plain code between them. The models only read a free-text answer and write the next question. Everything about flow (scoring services, deciding when to stop, breaking loops) is ordinary Python that is unit-tested and logged.

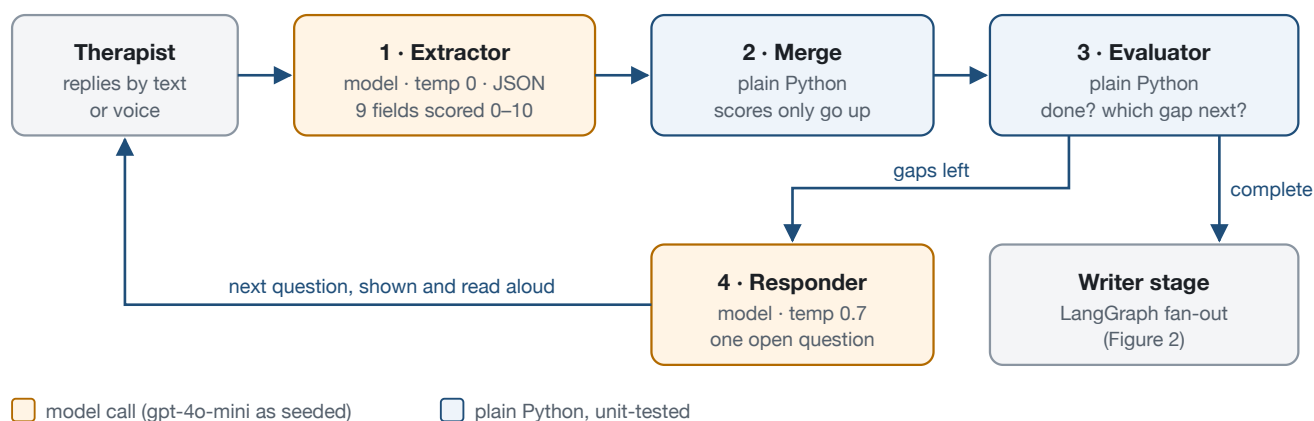


Figure 1. One conversation turn. Two model calls per turn; the decisions in between are code.

- 1. Extractor** (temperature 0, JSON mode). It gets the reply, the bot's previous question, a compact copy of the current state and the field definitions. The previous question was added so the model knows which field the person is answering; without it, answers were filed under the wrong field. It returns new snippets per field, a score for every field, merged text, a list of services and a day-by-day schedule. The prompt tells it to check each reply against all nine fields, not only the one asked about, and to tell apart orientations (CBT, psychodynamic), specialisms (trauma, couples) and service types (assessment, counselling).
- 2. Merge** (code). Scores can only go up. Services are matched by name with a fuzzy ratio of 0.6, so a service mentioned twice in slightly different words stays one service. Schedules overwrite day by day. Services and schedule are scored by counting rather than by the model: one named service scores 7, two score 9, three or more 10; the schedule scores 5 plus one per day with hours.

3. **Evaluator** (code). The chat can stop when `C_what_they_do` is at 7 or more and either (a) there is at least one service, one should-field at 7 and another at 6, or (b) the practitioner skipped services and three should-fields are at 7. Otherwise it returns the fields under 7 in priority order and notes which fields crossed 7 this turn.
4. **Responder** (temperature 0.7). It gets what is known, what just improved, what is missing and the last five turns, and writes the next question. Rules: one open question in one or two sentences, no summary of what the person said, no yes/no questions, never repeat a question, services first, then schedule, then profile gaps. Replies are read aloud by text-to-speech, so the prompt also swaps out Romanian words the voice mispronounces.

2.3 Rules that keep the conversation short

- **Minimum 4 turns**, even if the data is complete sooner, so the texts have some detail to work with.
- **Wrap up at turn 9** with whatever is there. The writers fill thin fields from domain knowledge and flag them.
- **Anti-loop**. If the same field is the target two turns running, it is accepted at 7 when it already scores 5 or 6, or dropped when it scores lower, and the chat moves on.
- **One extra question** when the required fields are done but two or more nice-to-have fields are still empty.
- **Skip** by button, or after turn 8 by typing *gata*, *next* and similar. The skip is refused, with a list of what is missing, if the must-fields are not covered.

2.4 Writer stage

When the chat ends, a LangGraph `StateGraph` drafts the texts. It has seven profile nodes (title, subtitle, short description, full description, team description, team expertise, approach) and three nodes per service (subtitle, short and full description). All of them start from `START` together and meet at a `finalize` node, which saves through the v1 save functions. Each node has its own prompt and gets only the fields it needs; a service node also gets every snippet that names that service. A practitioner with one service gets ten writers; one with three services gets sixteen. Each node writes to its own state key, and service results and errors are combined with reducers (a deep merge and list concatenation), so no node waits for another.

Each text comes back with a *weak* flag. It is set when any source field scored under 5, or, for a service text, when the chat never mentioned that service. Weak texts are highlighted on the review form so the practitioner knows what to check first.

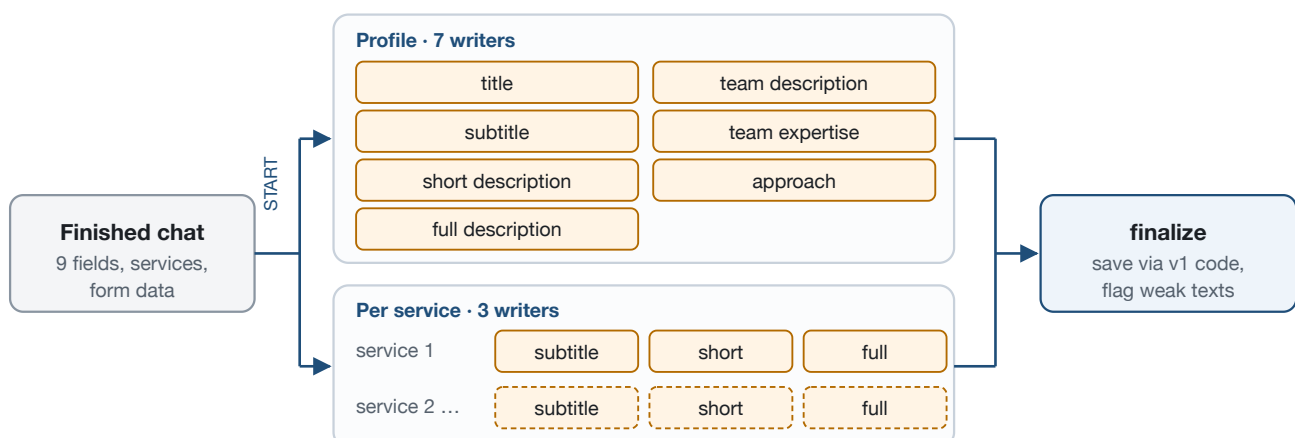


Figure 2. Writer stage. Every writer starts at once and they meet at `finalize`. Seven profile writers plus three per service: ten writers for one service, sixteen for three.

2.5 Prompts and models are data, not code

Every prompt and UI string is a database row (`V2Prompt`) keyed by slug and language, with a version number, an active flag and an activation time. A prompt can be edited in Django admin, and the logs from the period it

was live can be matched to that version. Model, temperature and token limit are stored per role (extractor, responder, generator) in `V2ModelConfig`; all three are seeded with gpt-4o-mini. Logs mask email addresses and phone numbers.

2.6 How it was tested

The v2 app has about 150 unit tests for the merge, scoring, stop and save logic. Conversation quality was tested with simulated practitioners, three ways:

- **Scripted browser runs** (Playwright). Each persona's answers are keyed to phrases in the bot's question, and the script clicks through the real UI. 20 personas.
- **Full-flow runs**. 15 personas through chat, services, tags, generation and review; 13 finished (two crashed when the browser closed during a dialog).
- **LLM-simulated users** (`simulate_onboarding` command). gpt-4o-mini plays a terse, verbose, bored, confused or careless-typing practitioner against the real pipeline, with real database sessions.

Table 2. Problems found by the 15 March 2026 full-flow runs, and the fixes.

Problem	Seen in	Fix
Plural “we” in texts for solo practitioners	13 / 13	Singular-voice rule added to five writer prompts
Year founded always empty	13 / 13	Key mismatch (<code>year</code> vs <code>year_founded</code>) fixed in the generator
Irrelevant tags, e.g. occupational therapy suggested to nutritionists	~8 / 13	Domain stop words added to the tag suggester
CBT named in texts when the person never mentioned it	5 / 13	“Do not assume methodology” added to the writer prompts

Re-runs for a nutritionist, a speech therapist and a psychodynamic therapist came back clean on all four. Average lengths across the 13 profiles: short description 615 characters, full description 2,412. The LLM-simulated runs found one more thing: the simulator let two conversations end at turns 2 and 3, under the 4-turn minimum. The simulator copies the view's stop logic instead of calling it, and the copy had drifted. The live view was correct.

3. prompt-audit: a harness for tuning prompts

3.1 The problem

A unit test tells you a view returned 200. It does not tell you that a prompt edit made the bot start inventing prices. prompt-audit tests behaviour: a fixed set of scenarios goes through the real pipeline, an LLM judge scores each answer against written criteria, and the scores are compared with earlier runs. It is built as two Claude Code skills, which run the process with the developer, and about 640 lines of Python, which do the mechanical parts.

3.2 Terms

- **Endpoint**: a Python entry point that calls an LLM.
- **Profile**: one way of testing an endpoint, as one role in one language (e.g. `generalist-en`). It holds a constitution (purpose, manner, a role description, expected behaviours, a risk surface), where every prompt lives, and a pinned judge model and judge-prompt version.
- **Scenario**: one YAML test: the input, the behaviours the answer must show, what it must not do, tags for what it stresses, and notes for the judge.
- **Report**: one dated run, with full snapshots of every prompt. **Baseline** and **drift**: per-scenario scores carried from run to run.

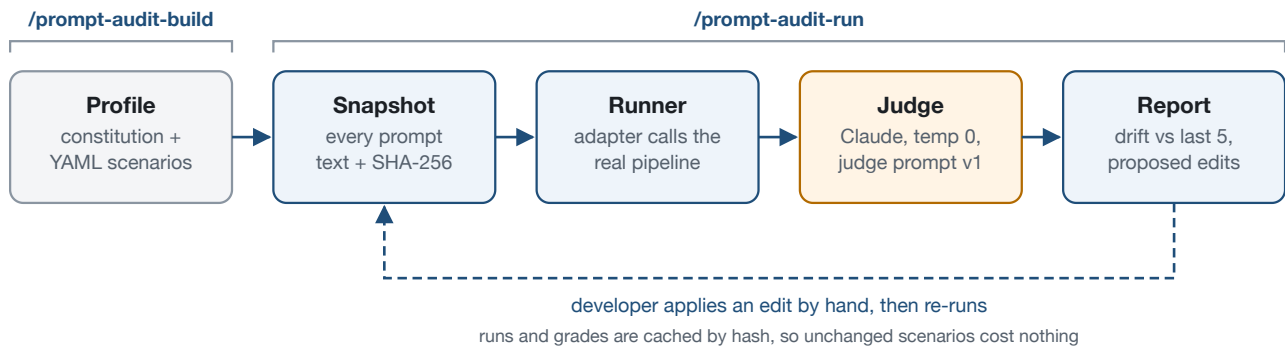


Figure 3. The harness. The build skill writes the profile; the run skill does the rest. Nothing is ever applied to a prompt automatically.

3.3 /prompt-audit-build

This skill only writes files; it never runs a scenario. It reads the endpoint from its entry function down to every LLM call and lists every prompt source it finds: database rows, files, constants, tool docstrings (the model reads them), Pydantic field descriptions, middleware, background tasks, webhook handlers, and fallback prompts hidden in error branches. It then shows an 18-item checklist of places it may have missed and waits for an answer to each one. It drafts the constitution one section at a time for approval and never fills a field on its own.

Scenarios come in two modes. *Curated* proposes 8 to 12, reviewed one at a time. *Bulk* proposes 20 to 50, approved by category, for broad first coverage of a new agent. The target mix is about 25% golden path, 20% edge cases, 20% refusals, 15% tool use, 10% known failure modes and 10% odd input (empty, very long, emoji only, injection-shaped). Bulk scenarios carry their batch date, so the ones that always score 5 or never move can be pruned later. A clone-translate mode copies a profile into another language and flags refusal and tone scenarios for a cultural check.

3.4 /prompt-audit-run

1. Load and validate the profile and its active scenarios.
2. Snapshot every prompt source before running anything: database rows, files and constants, each with a SHA-256.
3. Show the scenario list and estimated cost. No paid call runs until the developer says go.
4. Run each scenario through the endpoint's adapter against the real pipeline under test settings, and record the reply, tool calls, latency and tokens. A failing scenario is recorded as an error and the run continues.
5. Grade each answer with the judge.
6. Score each prompt from the scenarios that stress it.
7. Build a drift table over the last five reports. A drop of 2 points or more is a hard alert. A change of judge model or judge prompt is flagged, because scores across that line are not comparable.
8. For any prompt under 4, propose a diff with the reason and a confidence. It is never applied automatically.
9. Save the report with appendices: the profile, the full text and hash of every prompt, the judge prompt and the cost. The report is the rollback point. The baseline moves forward on its own only when there were no hard alerts.

3.5 The code

- `runner.py` loads a profile and a scenario and calls `adapters/<endpoint>.py`. The adapter's whole contract is `call(scenario, profile)`, returning the reply, tool calls, tokens and metadata. Results are cached under a hash of the scenario, the combined prompt hash and the model config, so a re-run with nothing changed is free and any prompt edit forces a fresh call.
- `snapshot.py` turns `db_row`, `file` and `code_constant` sources into text plus SHA-256.

- `judge.py` builds the grading message (constitution, scenario, reply, latency, tool calls) and calls Claude at temperature 0 with the pinned judge prompt. Its cache key is the reply, the criteria, the judge model and the judge-prompt version.
- `judge_prompts/v1.md` gives one 1–5 score over six factors: accuracy, helpfulness, speed, persona fit, no invention, and forward motion. Hard caps apply first: a must-not violation or any invented fact caps the score at 2, the wrong language caps it at 1, and a reply over 10 seconds caps it at 3. Every failure is tagged with its factor (`accuracy:` , `invention:` , `speed:` ...), so a report can show which kind of failure increased.

3.6 Case study: Sonia, 13 May 2026

Sonia is the shop assistant behind FormAI, DBM AI's chat widget for online shops. Its first profile, `generalist-en`, runs 15 bulk-generated scenarios against a demo WooCommerce shop with 51 products priced in euros. It was run five times in one day.

Table 3. Five audit runs of Sonia, `generalist-en`. Mean score out of 5.

Run	Mean	What changed	What the audit showed
R1	4.20	Baseline	Prices without a currency in 6 of 15 scenarios; three product searches over 5 s
R2	4.33	Currency rule added to the answer prompt; per-step timers added	The rule had no effect. Timings showed the answer model never saw the shop's currency; only the router did. The time went on the two model calls, not on WooCommerce.
R3	4.27	Answer prompt cut from 3,348 to 1,977 characters; router moved to gpt-5.4-mini; currency tag passed to the answer model	Personality switching fell from 5 to 2 (the router passed bad parameters). Mean latency rose 24%.
R4	4.20	Router moved back to gpt-4.1-mini	Personality switching back to 5. Latency varied run to run.
R5	4.47	Code post-processor adds the shop's currency to bare prices	Currency on 15 of 15 prices, up from 0% in R1 and R2

Three things came out of it. The router change was meant to make Sonia faster; it made her slower and broke a tool, and without fixed scenarios it would have shipped. The prompt rule for currency was the wrong fix: the model it was written for could not see the data it referred to, and once it could, gpt-4.1-mini still followed it only some of the time, so a few lines of deterministic code replaced it. And one run is not enough to judge latency.

4. Limitations

- **The judge was not independent.** In these five runs the grading was done by Claude Opus 4.7 inside the same Claude Code session, not by `judge.py` with its pinned Sonnet 4.6 at temperature 0. That session also proposed the fixes, so the grader was not separate from the change, and the scores cannot be reproduced exactly. Future runs should call `judge.py` as written.
- **One run per configuration.** Latency varies enough that each configuration needs three to five runs.
- **Small, unchecked suite.** 15 bulk-generated scenarios, not yet pruned. The scores are model judgments that no person has compared with human ratings.
- **The harness has only been used on Sonia.** The onboarding agent was tuned with persona simulations, which find problems but give no score to track. Bringing onboarding into prompt-audit needs a multi-turn adapter (the runner sends one message per scenario today). The `V2Prompt` rows already fit the `db_row` snapshot kind.
- **The simulator duplicates the view's stop logic** and has already produced one false result. Both should call one shared function.

5. Summary

Both systems split the work the same way: the model does the language, and code you can test makes the decisions. In onboarding, that split is what lets a cheap, non-reasoning model run a goal-driven conversation: two narrow model calls per turn, plain Python for scoring, stopping and loop-breaking, and a parallel fan-out for the writing. The Sonia runs show the same move from the other side: when gpt-4.1-mini would not follow a prompt rule reliably, a few lines of code fixed what the prompt could not. In prompt-audit the split means fixed scenarios, hashed prompt snapshots and a pinned judge, so a prompt change shows up as a number before it reaches users.

Appendix. Where the code lives.

Path	What it does
onboarding/v2/ingredients.py	The nine field definitions, priorities and the threshold of 7
onboarding/v2/extractor.py, responder.py	The two model calls per turn
onboarding/v2/evaluation.py	Merge, scoring of services and schedule, stop rule, next target
onboarding/v2/views.py	Turn control: minimum turns, wrap-up at 9, anti-loop, skip
onboarding/v2/generator.py	LangGraph writer graph and save
onboarding/v2/management/commands/simulate_onboarding.py	LLM-simulated practitioners
tests/prompt-audit/_runner/	Runner, snapshot, judge, adapters, versioned judge prompts
~/.claude/skills/prompt-audit-build, prompt-audit-run	The two Claude Code skills