B E L A

BELA — A SMALL AI WITH BIG AMBITIONS

An app-scoped, self-training agent engine that runs on a $4 CPU VPS.

Version 0.0.1 · MIT · TypeScript · github.com/rohanzaki/bela

---

1. THE PROBLEM

The standard advice for "add AI to my app" is to call a hosted LLM. For a small business application — a hotel front desk, a clinic scheduler, a shop's order screen — that advice fails on four counts at once.

Cost. Per-token billing scales with usage, not with value. A front-desk clerk typing forty short commands a day generates real recurring cost for a product whose whole margin lives in a $10/month VPS.

Privacy. Guest names, phone numbers, CNIC/ID data, order history — the exact fields a command touches are the ones you least want leaving the building. In many markets that is not a preference, it is a compliance requirement.

Reliability. An LLM that is up 99.9% of the time is down when the internet at the property is down, which for a lot of the world is a weekly event. An app feature that stops existing when a cable is cut is not a feature staff will build habits around.

Fit. A general-purpose model knows the entire internet except the one thing you need: *your* rooms, *your* menu, *your* staff's shorthand. You close that gap with prompt engineering, retrieval, and tool schemas — and you are still one model update away from a behaviour change you did not ask for.

There is a second, quieter problem. Most "AI agent" stacks will happily guess on a write. A model that is 95% right is a model that cancels the wrong guest's booking one time in twenty. For read-only chat that is an annoyance. For an action executor it is unacceptable.

2. THE BELA APPROACH

Bela is not a chatbot. It is an **action executor**: it reads one short command, resolves it against your app's real data, calls a function *you* registered, and replies with one short line. It does no open-ended conversation and it knows nothing outside your application.

Three ideas do the work.

2.1 THE MODEL IS BORN FROM YOUR APP

Bela ships with **zero domain knowledge**. There is no built-in concept of "room", "menu item", or "patient" anywhere in the codebase. You declare your entities and tools in one config file; Bela's training-data generator expands each tool's utterance templates against your *live* entity rows and a synonym table into thousands of labelled examples, and trains a small intent classifier and slot tagger from scratch — on your machine, in about a second on one CPU core.

The result is a model that has no existence outside your app. Two projects running Bela never share a model, a vocabulary, or a mistake. There is no pretrained LLM in the loop, no external API call, and nothing to leak.

2.2 A SMALL INDEPENDENT MIND

The architecture is deliberately shaped like an insect's nervous system rather than a brain. Local reflexes are guaranteed and always present; a big brain is optional and pluggable.

The reflex layer is deterministic: utterance-template matching, fuzzy entity resolution against a synced knowledge base, and dedicated extractors for dates (chrono-node), numbers, and identifiers. It is fast, auditable, and it either matches or it honestly does not.

Behind it sits the self-trained neural fallback, engaged only when no template matches at all — it catches the paraphrases you never thought to write down ("get karahi in room 4" against templates that only ever said "order").

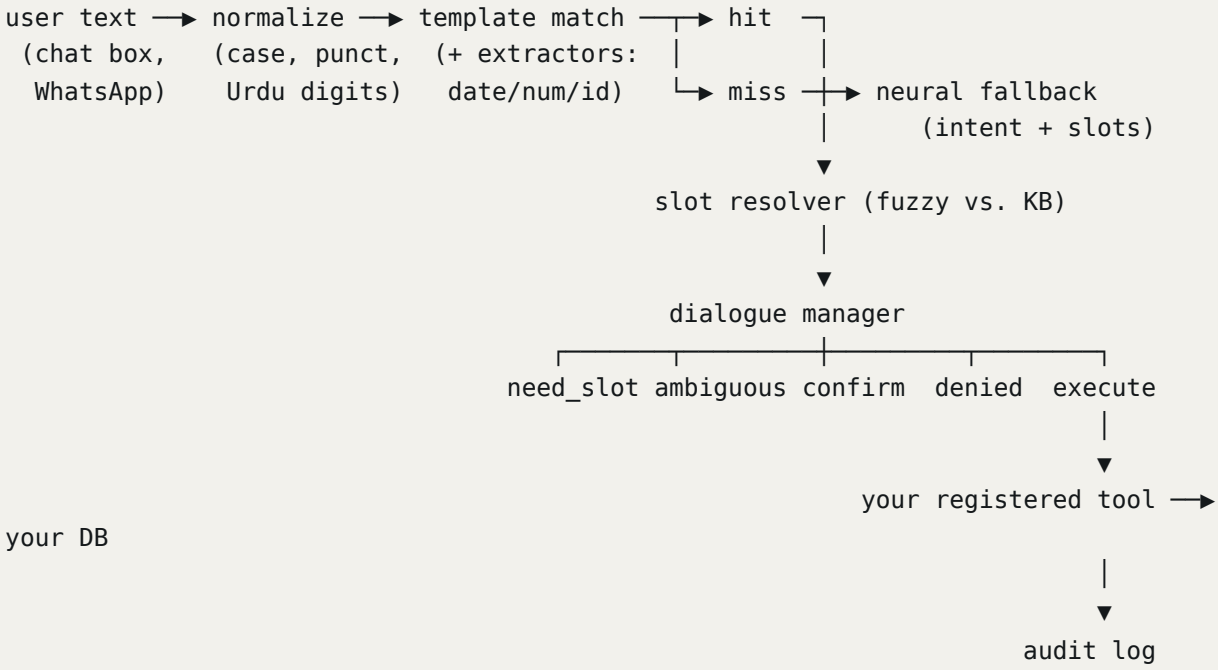
Behind *that* is a socket, not a dependency. The `FallbackFn` interface that the neural model plugs into is the same one an LLM adapter will plug into (roadmap, §7). Cut the head off and the insect keeps walking: if the LLM is unreachable, every trained command still works locally.

2.3 SAFETY IS STRUCTURAL, NOT PROMPTED

Bela cannot touch your database. It cannot write SQL. The only things it can do are call functions you explicitly registered, with arguments it has resolved to real rows in your data. Your business rules stay in your app, where you can test them.

On top of that the pipeline enforces four gates, described in §5.

3. ARCHITECTURE



The packages map one-to-one onto that picture:

PACKAGE	RESPONSIBILITY
`@bela/core`	Registry, normalization, template matcher, extractors, fuzzy resolver, dialogue manager, sessions, roles, audit
`@bela/model`	Training-data generator, tokenizer/hashing, intent classifier, slot tagger, artifact save/load, correction store
`@bela/cli`	`bela init` / `bela train` / `bela eval`
`fixtures/hotel`	Reference integration used for end-to-end tests

The knowledge base is a periodic pull: each entity you declare has a `resolve()` that returns its current rows (a Prisma query, a SQL call, an API fetch — your code). `agent.sync()` refreshes it. A sync failure keeps the last good snapshot rather than degrading the agent.

The developer contract is one file:

```
export default defineAgent({
  entities: {
    Room:     { resolve: () => prisma.room.findMany(), match: ["number"] },
    MenuItem: { resolve: () => prisma.menuItem.findMany(), match: ["name",
"aliases"] },
  },
  tools: [
    tool("orderFood", {
      params: { qty: number({ default: 1 }), item: entity("MenuItem"), room:
entity("Room") },
      utterances: ["order {qty} {item} for room {room}", "{item} for room {room}"],
      run: ({ item, room, qty }) => api.post("/api/orders", { itemId: item.id,
roomId: room.id, qty }),
    }),
    tool("cancelBooking", {
      params: { guest: entity("Guest") },
      utterances: ["cancel {guest} booking"],
      confirm: true,
      roles: ["manager"],
      run: ({ guest }) => bookings.cancel(guest.id),
    }),
  ],
})
```

`defineAgent` validates at build time: duplicate tool names, references to undeclared entities, and placeholders that do not correspond to a declared param all throw immediately rather than failing at runtime in front of a user.

4. MEASURED RESULTS

All figures below were produced by the repository's own benchmark scripts (`scripts/bench.mjs` , `scripts/train-bench.mjs`) against the reference hotel fixture, on a single CPU core with no GPU.

WHAT	MEASURED
Average command latency	~0.3 ms (dev machine: 0.068 ms avg, p99 0.21 ms over 2,000 commands)

WHAT	MEASURED
Self-train time, full model	~1 second on 1 vCPU (0.5 s measured)
Intent accuracy, held-out set	99.3–99.6%
Slot exact-match	100% on the fixture config
Trained model artifact	~8 MB JSON
Process RSS	~100–160 MB, serving and training
Neural fallback inference	~1.4 ms
Test suite	113/113 passing, 23 files
Runtime dependencies	one (`chrono-node`)
Hosting requirement	CPU-only VPS, $4–12/month, 512 MB–2 GB RAM

Two things are worth underlining. First, the latency figure is for the deterministic path, which handles the overwhelming majority of real commands; the neural fallback costs about a millisecond more and runs only on a template miss. Second, one-second training is what makes correction-driven learning practical — retraining is a routine deploy step, not an event.

The CLI enforces quality as a gate rather than a report. `bela train` runs the training, prints `intentAcc` / `slotExact` / `testCount`, checks them against thresholds (defaults `intentAcc >= 0.9`, `slotExact >= 0.8`), and **refuses to save the artifact** if they are not met. A config change that quietly wrecks model quality cannot ship.

5. THE SAFETY MODEL

Four gates sit between a parsed command and a mutation.

Never guess on writes. Low confidence, an unresolved required slot, or an entity that matched two rows (two guests named John) produces a question, not a best guess. `need_slot` and `ambiguous` outcomes carry the follow-up text; the per-user session remembers what was already filled in, so the user answers one short question rather than retyping the command.

Enforced confirmation. A tool marked `confirm: true` *never* executes on a fresh match. It returns a `confirm` outcome echoing the resolved arguments and waits for an explicit yes or no from the same user — in English or

Roman-Urdu (`haan` , `han` , `ji` / `nahi` , `nah`). There is no code path that skips this.

Role gates. A tool with a `roles` list checks the caller's role, supplied by your app per request, and returns a `denied` outcome instead of running. The tool never executes and nothing is mutated.

Audit trail. An optional `audit` callback fires on every `executed` , `error` , `denied` , and `cancelled` outcome with the user, raw input, tool, resolved arguments, and timestamp. If your audit callback throws, the pipeline swallows it — a broken logger cannot break command handling.

Around those: tool handler exceptions are caught and templated into a safe user-facing message (your error text never leaks to the user); sessions expire on a TTL so stale follow-ups die quietly; a retrain that fails leaves the previous artifact serving; and a corrupted or hand-edited model file fails fast at `loadModel` with a clear error instead of crashing deep inside prediction.

Unmatched input gets an `unknown` outcome whose reply lists what the agent *can* do, built from each tool's first utterance. Never a guess, never chit-chat.

6. AN HONEST COMPARISON

Bela is not a better LLM. It is a different shape of tool, and it loses badly outside its shape.

	BELA	CLOUD-LLM AGENT
Marginal cost per command	zero	per-token
Infrastructure	$4–12/mo CPU VPS	API account, or GPU
Works offline	yes	no
Data leaves your server	never	every request
Latency	sub-millisecond	300 ms – several seconds
Setup effort	write a config, train, wire it in	write prompts and tool schemas
Open-ended conversation	**no**	yes
Reasoning over novel situations	**no**	yes
Handles phrasings nobody anticipated	only via retrain on corrections	often, zero-shot
Multi-step planning	**no**	yes

	BELA	CLOUD-LLM AGENT
Behaviour changes under you	never (artifact is yours)	on provider model updates
Determinism / auditability	full	partial

Read the bold rows carefully. If your users need to *converse*, reason about things outside the app, or chain several unplanned steps, use an LLM — Bela will not get there, and the roadmap's LLM adapter exists precisely so you do not have to choose. If your users need to *fire a known action fast, safely, cheaply, and offline*, Bela is the smaller and stronger tool.

The honest weakness is coverage. Bela's understanding is bounded by the utterance templates you write plus what the generator and neural fallback can extrapolate from them. It closes that gap by learning rather than guessing: mis-parsed phrasings are captured through `CorrectionStore` and folded into the next retrain, so the model converges on how *your* staff actually talk — including their Roman-Urdu shorthand — instead of you hand-editing templates forever. Corrections pass the same accuracy gate, so a bad correction cannot degrade the live model.

7. ROADMAP

Memory and history. A built-in activity store using `node:sqlite` (built into Node 22+, so still zero new dependencies) turns the audit hook into queryable history: "what did you do today?", per-user activity, correction dedup, undo candidates. Alongside it, a `bela-memory.md` file the user owns — readable, editable, portable to a fresh install. Its entries are *structured facts*, not fuzzy context: an alias ("room 4 = corner room") becomes a resolver synonym, a phrasing fact becomes a training correction. Memory is data, never instructions, and no LLM interprets it — so prompt-injection-shaped risks largely do not apply. Your AI's brain is a file you own.

LLM adapter, both directions. *Call-up*: on low-confidence input, optionally consult a configured LLM over any OpenAI-compatible endpoint. The LLM may only **suggest** a tool call; the suggestion still passes through Bela's resolver, role gate, confirmation, and audit trail. If the LLM is down, every trained command still works. *Called-down*: expose Bela as an MCP server so big-LLM agent stacks can invoke your app's actions **through** Bela's safety gates — Bela as the hands layer for any agent.

This also unlocks **targeted distillation**: ask a large model once for exactly what your project needs — say, the Roman-Urdu phrasings your staff use for "checkout" — fold the answers in as training data, and the knowledge lives locally, forever, at zero marginal cost. You rent the big model for an afternoon instead of paying it per command for years.

Agent linking. The core is transport-agnostic, so another Bela is just an authenticated caller with a role — meaning roles, confirmations, and audit apply to a foreign agent for free. Needs shared-key auth, loop prevention,

and provenance in audit entries ("by hotel-agent on behalf of desk1").

Sidecar for non-Node hosts. PHP/WordPress, Laravel, and Django are an explicit roadmap item: Bela runs as its own small process, tools are declared as REST endpoints in a JSON/YAML config, and entity resolvers become read-only HTTP endpoints. Same engine, HTTP transport instead of in-process function calls — no redesign, just not built yet.

8. FAQ

Does it call an LLM API? No. No external AI calls and no per-token cost. The core is deterministic TypeScript; the optional fallback is a model you train yourself, from scratch, on your own config and data.

Can it write to my database directly? No, and it never will. It can only call tools you registered. Your `run` handlers are your own functions.

What if a command matches nothing? You get an `unknown` outcome listing what the agent can do. Never a guess.

Do I need a GPU? No. Training and inference are both CPU-only, single core, and finish in about a second.

What languages? Command structure is English-shaped; entity *values* — guest names, menu items, room labels — work in any script, and Urdu digits normalize to ASCII automatically. Roman-Urdu confirmations (`haan` / `nahi`) are built in, and Urdu/Roman-Urdu phrasings can be added as utterance templates or learned from corrections.

Is it production-ready? It is v0.0.1 and honest about it. The deterministic engine, self-trained neural fallback, dialogue safety, and CLI all work and are covered by 113 passing tests including end-to-end fixtures. The first production integration is live: Bela runs inside a deployed hotel & restaurant management system, trained on its real data (99.5% held-out intent accuracy), executing staff commands — including Roman-Urdu phrasings — through the app's own permission-checked functions, with WhatsApp delivery verified end-to-end. The npm release is the current phase of work.

What is the licence? MIT. The core stays MIT.

Bela — the smallest useful AI model people can run to operate their own application. Nothing more, nothing less.