

Declarative Dialog Flows for Multi-Tenant Conversational Agents: Design Lessons from ARC

Vivek Sinha
Shielva

Abstract

ARC is a multi-tenant platform on which small businesses build conversational agents for chat, messaging and telephone. Authors draw an agent as a typed node graph covering conversation, logic, data handling and external capabilities, and a server-side runtime executes it together with intent classification, document retrieval and a catalogue of more than 200 integration connectors. This report describes three design choices and the problems that motivated them. First, the flow runtime is a pure step function that advances one user turn and expresses every side effect as an explicit request, so the same graph runs identically in an authoring simulator and in production. Second, agents installed from a shared template now follow that template automatically, using a content-hash version and a conditional write so an upgrade happens exactly once and preserves each business's own integration choices. Third, the variables that automations pass into conversations are governed by one declaration registry checked by a parity test, after the same declaration drift recurred three times and once caused an agent to speak template syntax to a caller. We report the operational measurements we have and state what remains unvalidated.

Keywords: conversational agents, dialog management, low-code platforms, multi-tenancy, software architecture, retrieval-augmented generation

1. Introduction

Clinics, property agencies and similar small businesses want agents that answer from their own information, collect structured details, book appointments and hand off to staff on whichever channel a customer uses. Prompting a large language model alone is hard to audit and hard for a business to steer; hand-written state machines are hard for non-engineers to change. Visual dialog flows sit between the two, and both open frameworks such as Rasa [1] and commercial builders occupy this space. ARC combines an author-drawn flow with language-model steps, retrieval-augmented generation [2] and integration actions, and serves many businesses from one deployment.

This is an experience report rather than a controlled study. Its contribution is three design patterns, each adopted after a concrete failure, that we believe transfer to other agent-building platforms.

2. System Overview

An agent consists of a flow graph, a persona, optional knowledge bases, and opt-in links to business-wide intents, signal rules and action definitions. Four cooperating components serve it:

- **Authoring console** with a graph editor and an in-browser simulator.
- **Flow runtime**, which executes the graph one user turn at a time.
- **Understanding layer**, a cascade from normalisation and explicit rules through cached statistical matching to entity extraction, returning one of a small set of next actions (execute, ask for information, escalate, unmatched).
- **Retrieval and integration layer**: per-business knowledge collections searched with a hybrid of dense vectors and BM25 [3] fused by reciprocal rank [4], and a connector runtime that loads each integration as an independently versioned plugin package.

Every request is attributed to exactly one business by the platform's identity layer, never by data supplied in the request itself, and every stored record is scoped to that business.

Node kinds. The editor offers about thirty node kinds in four families: conversational (message, ask, classify, answer, hand-off), logic (condition, set, loop, wait, route, sub-flow), data (filter, sort, map) and capabilities (messaging, mail, calendar, CRM, custom actions). Edges connect named ports, so a classification node's cases are simply its outgoing ports.

3. The Runtime as a Pure Step Function

The runtime exposes one operation: given a flow, the current conversation state and a user message, return the new state and the output for this turn. It reads and writes no storage and makes no network calls. When a node needs the outside world (classify a message, validate an answer, call an integration, run a language-model agent) the runtime emits a typed request and stops. An orchestrator performs the work and re-invokes the same turn with the result.

Three benefits followed. The authoring simulator and the server share one semantics, so an author tests the real behaviour before publishing. Side effects are visible in the protocol rather than hidden in node code, which made several classes of defect in Section 6 easy to see. And a turn can be replayed safely. Two small rules keep the walk well-behaved: a per-turn bound on visited nodes stops cyclic graphs, and sub-flows are expanded before the walk begins. Condition nodes take the first true case, and a case with no test acts as the default.

Conversation state, including loop and pacing positions, is held in a short-lived cache, so a paced batch of external writes resumes where it stopped instead of repeating work. Long-term facts an agent learns about a customer are consolidated separately with their own retention period.

4. Templates That Installed Agents Follow

A template packages a persona, intents, signal rules, decision rules, capabilities, suggested integrations and a flow. Installation follows a fixed order: the agent, then its understanding configuration, then the flow last, so that classification cases never refer to intents that do not yet exist. A capability node is bound to an integration automatically only when the business has exactly one candidate; otherwise the choice is left to the author.

Originally installation was a one-time copy. When a template improved, installed agents did not: at one point live agents ran a 23-node flow while the current template had 30. A manual re-apply existed but was easy to run at the wrong moment, and once restored an older template during a staged rollout.

We replaced the copy with a link. A linked agent records its template and a version equal to a truncated SHA-256 over the template flow's canonical JSON. When the agent's flow is read and the version differs, the reader rebuilds the flow, carries each business's integration choices across by node identifier, keeps the replaced graph in history, and writes on the condition that the stored version is still the old one. Readers running concurrently therefore upgrade an agent exactly once. A linked flow refuses direct edits; unlinking is the explicit act that makes a flow the business's own, and linking is a preview until confirmed.

The design trades flexibility for consistency: a business that wants a custom flow must unlink and then stops receiving improvements. We considered three-way merges of template changes into customised flows and rejected them as too unpredictable for non-technical authors.

5. Declarations as the Contract

Automations such as campaigns, screening calls and booking workflows start conversations with call variables that authors reference as `{{name}}`. Substitution leaves an unknown variable untouched so that the mistake stays detectable rather than becoming a silent blank. The same mistake nevertheless recurred three times in different automations: one offered authors 9 variables while sending 20; another sent a value it never declared; a third declared values nothing sent, and a voice agent read the braces aloud.

Each automation now declares the variables it sends, with a sample payload, in one registry, and a parity test compares every declaration with what its sender actually emits. Values that must not appear in the authoring menu are declared as hidden rather than omitted. Values collected during the conversation take precedence over call variables, and empty scripted lines are dropped rather than spoken. The same principle now governs configuration screens: an automation declares its fields and limits and one shared library renders them, replacing a hand-written panel per automation.

6. Defects That Shaped the Design

- **One slot for many effects.** Capability nodes originally shared a single pending-request slot, so in a calendar, CRM, mail sequence only the last ran. Each now pauses and resumes on its own.
- **Two names for one contract.** The editor already defined the fields a capability node carries, but the first runtime implementation invented its own names, and the nodes silently did nothing. The editor's contract is now the single source.
- **Per-process notifications.** Live-update events were tracked per server process, so a stale editor's autosave overwrote a 15-node flow with a 1-node flow. Notifications moved to a shared channel.
- **Experiments pinned by position.** A/B variants referred to positions in a capped version history, which shift as history grows; they are now pinned by content.
- **Invented facts.** When a booking capability was unavailable to an agent, a language-model step filled the gap with plausible appointment times. Capability results now carry an explicit, author-visible sentence for the unavailable case, so the agent says so instead of improvising.

7. Measurements

Table 1. Operational measurements available to us.

What	Result
Connector packaging	213 of 213 connector packages built; all loaded in one process
Shorter spoken-answer prompt	reply 362 to 186 characters; about half the speech-synthesis and output tokens; 3.7 s faster
Agentic turn (three model calls) vs plain turn	7,164 ms observed vs about 750 ms
Long-term memory recall	one embedding per conversation instead of one per turn

These are measurements on individual runs, not controlled experiments. Telephony latency is analysed in a companion report [5].

8. Limitations

- Conversation state lives in a short-lived cache and is not durable beyond it.
- Token streaming from retrieval is available only for agents without tools.
- A/B experiments do not yet receive satisfaction feedback from the customer-facing widget.
- Some node kinds available in the editor have no runtime behaviour yet.
- We have no user study of authoring effort or comparison against other builders.

9. Conclusion

Three ideas carried most of the weight: keep the flow runtime pure and make effects explicit; let installed agents follow their template through versioned, conditional upgrades instead of one-time copies; and treat declarations as the contract between the producers and consumers of variables, enforced by tests. Each turned a class of silent failures into visible ones.

References

1. T. Bocklisch, J. Faulkner, N. Pawlowski, A. Nichol. Rasa: Open source language understanding and dialogue management. arXiv:1712.05181, 2017.
2. P. Lewis et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems* 33, 2020.
3. S. Robertson, H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval* 3(4), 2009.
4. G. V. Cormack, C. L. A. Clarke, S. Buettcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. In *Proc. SIGIR*, 2009.

5. V. Sinha. Building a production telephony voice agent on a cascaded pipeline: An experience report. Zenodo, 2026.
doi:10.5281/zenodo.23243533.