

# Karna: A Chat-Native, Multi-Channel Architecture for Personal AI Chief-of-Staff Agents

Mukunda Rao Katta  
Independent Researcher

## Abstract

Personal AI assistants are moving from single-window chat interfaces toward systems that must operate across messaging channels, web dashboards, mobile clients, voice notes, workflow triggers, and long-running memory. This paper presents Karna, a self-hosted TypeScript monorepo that explores a chat-native architecture for a personal AI chief-of-staff agent. The system is organized around a gateway, an agent runtime, shared protocol types, channel adapters, web and mobile surfaces, a CLI, memory/session modules, workflow hooks, observability surfaces, and deployment artifacts. Rather than proposing a new model or benchmark, the contribution is a concrete architecture pattern for building assistants that can preserve context across channels while keeping tool access, session coordination, deployment, and operator visibility explicit. Repository inspection identifies 13 channel packages, 485 TypeScript or TSX source files, 84 test files, Docker and Kubernetes deployment artifacts, and separate web, mobile, cloud, and command-line applications. The paper describes the design goals, repository structure, channel abstraction, operational safety considerations, and research questions that emerge from using chat as the primary interface for agentic work.

## 1. Introduction

Many personal-assistant prototypes begin as a chat box connected to a large language model. That starting point is useful, but it does not match how people actually coordinate work. Conversations arrive through messaging platforms, voice notes, meetings, reminders, browser tasks, mobile notifications, and dashboards. A useful assistant therefore needs more than a prompt. It needs an operating surface that can receive events from multiple channels, preserve context, route work through tools, remember commitments, and expose enough observability for the user to trust it.

Karna was built as an implementation-oriented exploration of that problem. It treats the assistant as a channel-native system rather than a single application. The project includes a gateway, agent runtime, shared schemas, channel packages, web and mobile clients, a command-line interface, memory and session modules, plugin interfaces, deployment manifests, and tests. The central question is: what repository and runtime architecture helps a personal AI assistant remain useful across many communication surfaces without hiding important operational state from the user?

This paper documents Karna as a systems artifact. It does not claim that Karna outperforms other assistants or that it provides a complete evaluation benchmark. Instead, it contributes an inspectable design pattern for chat-native personal agents and identifies the implementation boundaries that matter when moving from a toy assistant to a daily-use assistant stack.

## 2. Artifact Overview

The Karna repository is a TypeScript monorepo. The inspected version contains a root package configured for Node.js 20 or later, pnpm workspaces, Turborepo build orchestration, Vitest tests, and TypeScript compilation. The implementation includes 485 TypeScript or TSX source files and 84 test files, excluding dependency folders. The repository also contains Docker Compose, Docker, Kubernetes, Render, GitHub Actions, and static-site deployment artifacts.

The top-level organization separates responsibilities into the following areas:

- `agent`: the core runtime, including context, memory, tools, workflows, voice, sandboxing, feedback, retrieval, and orchestration modules.
- `gateway`: HTTP, WebSocket, routing, session, security, observability, integrations, webhooks, cron, and protocol handling.
- `apps/web`: the browser application and dashboard.
- `apps/mobile`: the mobile application surface.
- `apps/cli`: command-line workflows for local operation.
- `apps/cloud`: cloud-facing application package.
- `channels`: adapters for messaging and chat surfaces.
- `packages/shared`: shared protocol and session types.
- `packages/plugin-sdk`: plugin interface and examples.
- `packages/supabase`: database schema and migrations.
- `skills`: built-in and community skill definitions.
- `tests`: tests grouped across agent, gateway, channels, CLI, cloud, mobile, shared, and web surfaces.

This structure reflects a design choice: the assistant is not a single frontend connected to a model. It is an agent platform with separate surfaces and explicit adapter boundaries.

### 3. Design Goals

Karna is organized around five design goals.

First, the assistant should be chat-native. Messaging channels are not treated as secondary notifications. They are first-class event sources and response surfaces.

Second, memory and session state should be explicit. Long-running assistants need to know what is part of the current turn, what belongs to a session, and what should persist across future interactions.

Third, channels should be replaceable. Each integration should map external events into common protocol types without forcing the core agent runtime to know platform-specific details.

Fourth, operator visibility should be built into the system. A personal assistant that can act across tools needs logs, traces, health checks, analytics, and dashboard surfaces.

Fifth, deployment should be self-hostable. The repository includes Docker and Kubernetes artifacts so the assistant can be run under user-controlled infrastructure rather than only as a hosted service.

### 4. Channel Adapter Pattern

The `channels` directory includes packages for Discord, Google Chat, iMessage, IRC, LINE, Matrix, Signal, Slack, SMS, Teams, Telegram, Webchat, and WhatsApp. This gives Karna 13 channel packages in the inspected repository. The adapter pattern separates platform-specific concerns from the assistant runtime.

Each channel can be understood as a boundary that performs four tasks:

- Receive a platform event.

- Normalize that event into shared internal types.
- Forward the normalized event to the gateway or runtime.
- Deliver the assistant response back to the platform.

This pattern helps reduce coupling between the assistant core and external platforms. It also creates a clean place to implement channel-specific safety rules. For example, direct messages, group mentions, phone-number-based channels, and workspace channels can require different defaults for identity, consent, and reply behavior.

## 5. Gateway and Runtime Separation

Karna separates gateway responsibilities from runtime responsibilities. The gateway owns protocol handling, routing, session coordination, health, access checks, observability, webhooks, and integrations. The agent runtime owns context construction, memory access, tool use, workflows, skills, voice handling, retrieval, and orchestration.

This separation matters because personal assistants often fail when transport logic, model prompts, tool execution, and memory updates are mixed together. Keeping the gateway and runtime apart creates clearer places to test behavior. Gateway tests can focus on sessions, WebSockets, routes, and protocol flow. Runtime tests can focus on prompts, memory behavior, tools, and workflow decisions.

## 6. Memory and Session Considerations

Persistent assistants need memory, but memory can become risky if it is implicit. Karna's architecture makes memory and session modules visible in the repository structure. This makes it possible to reason about what state is used during a turn, what is retained, and what is exposed through the dashboard or channel replies.

For research purposes, the important point is not the specific memory implementation. The important point is that memory is treated as an operational subsystem. It should have types, tests, storage boundaries, safety checks, and deletion or correction paths. Chat-native assistants need these controls because users may disclose sensitive details across many surfaces.

## 7. Operational Safety

Karna's repository includes safety-relevant architecture surfaces: access modules, audit modules, session management, gateway security directories, channel separation, plugin boundaries, and deployment configuration. These pieces are necessary because personal assistants can cross trust boundaries quickly. A message from one channel may trigger a workflow in another system. A voice note may become a task. A group chat may contain instructions that should not apply to private memory.

The architecture therefore suggests several safety principles for chat-native assistants:

- Treat channel identity as part of the security model.
- Keep direct-message, group, and workspace defaults separate.
- Require explicit boundaries around tool execution.
- Store session and memory events in inspectable forms.
- Expose enough operational state for users to correct assistant behavior.

## 8. Evaluation Path

This paper does not present a benchmark score. Instead, it proposes an evaluation path for systems like Karna.

The first layer is repository-level evaluation: build success, type checking, unit tests, channel adapter tests, gateway protocol tests, and deployment manifest validation.

The second layer is interaction-level evaluation: replaying representative conversations, checking whether the assistant preserves task state, and verifying that channel-specific safety defaults hold.

The third layer is operational evaluation: measuring whether users can inspect sessions, trace actions, recover from wrong memory, and understand why a workflow fired.

The fourth layer is longitudinal evaluation: observing whether the assistant remains useful over repeated days of real tasks without accumulating stale, unsafe, or confusing context.

## 9. Discussion

Karna illustrates a shift from chatbots to assistant systems. In a chatbot, the central object is a message. In a personal chief-of-staff agent, the central object is an ongoing relationship between messages, tasks, memory, tools, and channels. This changes the architecture. The runtime must be testable. The gateway must be explicit. Channel adapters must be isolated. Memory must be inspectable. Deployment must be ordinary enough that a user or team can own it.

The repository also shows the cost of ambition. A multi-channel assistant has more moving parts than a single chat app. The engineering challenge is to keep boundaries small, typed, and testable while still supporting real surfaces where users already work.

## 10. Limitations

This work is an artifact report, not a controlled user study. The repository inspection provides implementation evidence, but it does not establish productivity gains, user satisfaction, or long-term retention. The channel packages indicate architectural breadth, but production readiness can vary across integrations. Future work should include replay-based evaluations, privacy reviews, deployment exercises, and longitudinal usage studies.

## 11. Conclusion

Karna provides a concrete architecture for a chat-native personal AI chief-of-staff assistant. Its monorepo structure separates channel adapters, gateway responsibilities, runtime behavior, applications, shared types, plugins, deployment, and tests. The paper argues that this separation is useful for moving personal assistants beyond single-window chat into multi-channel, memory-aware, self-hostable systems. The main contribution is an implementation-backed design pattern: make channels first-class, keep runtime and gateway responsibilities separate, make memory inspectable, and treat operator visibility as part of the assistant rather than an afterthought.

## References

Karna repository. <https://github.com/MukundaKatta/karna>

Model Context Protocol. <https://modelcontextprotocol.io/>

OpenTelemetry. <https://opentelemetry.io/>

Turborepo. <https://turbo.build/>

Vitest. <https://vitest.dev/>