Omnichannel Communication Architecture for Autonomous AI Agents

Surajbhan Satpathy and the Yoctotta Research Team

surajbhan.satpathy@yoctotta.com

Kaman AI Research · March 2026

Abstract

Deploying autonomous AI agents into production environments requires meeting users across a fragmented landscape of communication platforms—each governed by distinct protocols, authentication models, message-size constraints, and delivery semantics. We term this the *channel fragmentation problem*: the combinatorial complexity that arises when n agent capabilities must be surfaced uniformly across m heterogeneous platforms. This paper presents the omnichannel communication architecture of the Kaman platform, which currently supports 16 distinct channel types spanning messaging applications (WhatsApp, Telegram, Discord), team collaboration tools (Slack, Mattermost, Microsoft Teams), asynchronous channels (email, SMS), voice interfaces (Alexa), scheduled execution (cron), and generic webhook endpoints. We introduce a hub-and-spoke design centered on a *channel abstraction layer* that normalizes inbound messages into a universal envelope, routes them through the agent reasoning core, and delivers streaming responses back through platform-specific adapters—all while maintaining session continuity, enforcing per-sender access control, and supporting human-in-the-loop confirmation protocols. We discuss the design of a typed message segment protocol that separates content, suggestions, interrupts, and artifact delivery without exposing internal framing to external platforms. We further describe a hybrid deployment model in which channels may operate as in-process adapters or as independent microservices communicating via a message broker, enabling hot-reload and horizontal scaling. Empirical deployment data across production tenants demonstrates sub-second webhook-to-first-token latency for synchronous channels and reliable delivery across platforms with fundamentally incompatible streaming semantics.

1 Introduction

The proliferation of AI agents capable of autonomous reasoning, tool invocation, and multi-turn dialogue has introduced a new systems challenge: *where* should these agents be accessible? Enterprise users communicate through Slack and Microsoft Teams; customers prefer WhatsApp and Telegram; operational workflows demand email integration and scheduled execution; accessibility requirements increasingly call for voice interfaces. Each platform imposes its own constraints: Slack enforces three-second webhook response windows, Telegram caps messages at 4,096 characters, WhatsApp requires media uploads through a two-phase API, Alexa demands sub-seven-second total response times, and email operates on a fundamentally asynchronous send-receive paradigm.

A naive adapter pattern—in which each platform receives a bespoke integration with the agent core—fails for several reasons. First, it produces $O(n \times m)$ integration complexity as the number of agent capabilities and platforms grow independently. Second, session state management becomes fragmented: the same user interacting with the same agent through Slack and WhatsApp would see disjoint conversation histories. Third, cross-cutting concerns such as access control, credit-based usage gating, and human-in-the-loop tool confirmation must be reimplemented for each channel, creating maintenance burden and security risk.

This paper presents the communication architecture of the Kaman platform, which addresses the channel fragmentation problem through three contributions:

1. A **channel abstraction layer** that defines a minimal contract—`receive`, `process`, and `respond`—enabling new platforms to be integrated by implementing a single adapter class without modifying the agent core.
2. A **typed message segment protocol** that normalizes the diverse outputs of the agent reasoning engine (streaming text, tool confirma-

tion interrupts, file artifacts, suggested actions) into a uniform internal representation, with per-channel rendering strategies.

3. A **hybrid deployment model** supporting both monolithic in-process channels and independently deployed microservice channels communicating over a message broker, with runtime discovery and hot-reload semantics.

The architecture currently supports 16 channel types in production: WhatsApp Cloud API, WhatsApp via direct protocol (Baileys), Telegram, Slack, Mattermost, Discord, Microsoft Teams, email (SMTP/IMAP), Gmail API, Twilio SMS, Alexa Skills, Google Workspace, Odoo Events, generic HTTP webhooks, cron-scheduled execution, and a first-party web chat interface.

2 Design Principles

The architecture is governed by five design principles that emerged from iterative deployment experience.

2.1 Channel-Agnostic Agent Core

The agent reasoning engine must have zero knowledge of the delivery channel. Messages arrive as normalized text (or structured input for file and voice content); responses are emitted as a stream of typed segments. The agent never branches on channel identity. This ensures that adding a new channel requires no modifications to the reasoning pipeline.

2.2 Streaming-First Design

Large language model inference—routed through the platform's multi-provider abstraction layer [10]—produces tokens incrementally. The architecture assumes streaming as the default delivery mode, with buffering as a degradation strategy for platforms that cannot accept incremental updates. This is the inverse of the more common batch-first design in which streaming is retrofitted.

2.3 Session Continuity with Isolation

A single user interacting with multiple agents through the same channel must maintain separate conversation histories. Conversely, a single user interacting with the same agent through different channels may optionally share history. Session identifiers are therefore scoped as a composite of sender identity and downstream agent identity, preventing cross-agent session collisions while permitting cross-channel continuity where configured.

2.4 Graceful Degradation

Not all platforms support all agent capabilities. Rich artifacts (files, images, charts) may be delivered natively on platforms with media APIs and degraded to download links on text-only platforms. Human-in-the-loop confirmations use inline buttons on platforms that support them and fall back to text-based prompts elsewhere. The architecture never fails due to a capability mismatch; it degrades.

2.5 Security at the Channel Boundary

Each inbound message is verified at the channel adapter before entering the processing pipeline. Verification is platform-specific: HMAC-SHA256 for Slack and WhatsApp, Ed25519 for Discord, certificate-chain validation for Alexa, and shared-secret token verification for Telegram. Messages that fail verification are rejected before any downstream processing occurs.

3 Architecture Overview

The system follows a hub-and-spoke topology. The *hub* is the agent orchestration core—a stateful graph executor backed by persistent session checkpointing. The *spokes* are channel adapters, each responsible for translating between platform-specific protocols and the universal message envelope.

3.1 Three-Phase Message Lifecycle

Every interaction traverses three phases:

1. **Receive.** The channel adapter accepts an inbound event (HTTP webhook, WebSocket message, polled email, or scheduled trigger), verifies its authenticity, extracts the sender identity and message content, and constructs a normalized message envelope.
2. **Process.** The envelope is routed to the appropriate downstream resource—typically an agent, but potentially a workflow or a standalone function. The agent core processes the message, producing a stream of typed response segments.
3. **Respond.** Response segments are intercepted by a callback layer that classifies each segment (content, interrupt, artifact, error, suggestion) and routes it to the channel adapter's deliv-

ery method, which renders it in a platform-appropriate format.

3.2 Downstream Resource Routing

A channel instance is bound to a *downstream resource identifier* that specifies both the type and identity of the target: agent, application, workflow, or function. This binding is established at instance creation and allows the same channel type to serve different agents. The routing layer parses this identifier and dispatches accordingly, with agents receiving full conversational context and workflows receiving single-shot invocations.

3.3 Deployment Topology

The architecture supports two deployment modes simultaneously:

- **In-process adapters**: Channel implementations that run within the agent service process, suitable for webhook-based channels with low resource overhead.
- **Microservice channels**: Independently deployed processes that communicate with the agent core via a message broker, suitable for channels requiring persistent connections (e.g., direct WhatsApp protocol) or heavy resource usage.

Both modes coexist in production, with the routing layer attempting microservice delivery first and falling back to in-process handling transparently.

4 Channel Abstraction Layer

The abstraction layer defines the contract that every channel adapter must satisfy. We describe the contract, the adapter lifecycle, and the mechanisms for extending the system with new channels.

4.1 The Channel Contract

Every channel adapter implements three core operations:

- `receiveMessage(callback, message)`: Accepts a platform-specific inbound event, classifies it (text message, voice note, callback query, verification challenge), and initiates downstream processing.
- `messageBack(segment, sessionId)`: Receives a typed response segment from the agent core and delivers it to the appropriate recipient on the external platform.
- `sendOutboundMessage(message, target)`: Supports proactive messaging—agent-initiated communication outside the request-response cycle, used for scheduled notifications and workflow-triggered alerts.

Additionally, channels declare a static `configDefinition` that specifies the configuration fields required for instantiation (API tokens, signing secrets, access control modes), enabling the platform's management interface to render dynamic configuration forms without per-channel UI code.

4.2 Adapter Lifecycle

Channel adapters follow a defined lifecycle:

1. **Registration**: On system startup, the platform scans for channel implementations, invokes each adapter's static initialization method, and persists the resulting channel definition (identifier, human-readable name, description, configuration schema, setup instructions) to a registry.
2. **Instantiation**: Users create channel *instances* by selecting a channel type and providing configuration values. Each instance is bound to a specific downstream agent and user, receiving a unique instance identifier.
3. **Activation**: Active instances are started on system boot. For webhook-based channels, this is a no-op. For channels requiring persistent connections (direct messaging protocols, IMAP polling, cron scheduling), the startup process initializes the necessary long-lived resources.
4. **Hot-reload**: Configuration updates to active instances trigger teardown of the old instance and creation of a new one with updated parameters, without requiring system restart.

4.3 Adding a New Channel

Integrating a new platform requires implementing a single adapter class that extends the base channel contract. The adapter must:

1. Define its configuration schema via `configDefinition`.
2. Implement `receiveMessage` to parse platform-specific webhooks.
3. Implement `messageBack` to buffer and deliver responses.
4. Implement `sendOutboundMessage` for proactive messaging.

5. Provide a static `initialize` method returning the channel definition.

No modifications to the agent core, routing layer, or management interface are required. The new adapter is automatically discovered at startup and made available for instance creation.

5 Message Normalization Protocol

The diversity of agent outputs—streaming text, tool invocation requests, file artifacts, error conditions—requires a normalization protocol that preserves semantic type information while remaining transport-agnostic.

5.1 Typed Message Segments

The agent core emits a stream of typed segments, each carrying a semantic prefix that identifies its role in the conversation flow. We identify five segment types:

- **Content segments**: Incremental text output from the language model, delivered as the agent generates tokens.
- **Suggestion segments**: End-of-turn markers that signal the channel to flush its buffer and deliver the accumulated response to the user.
- **Interrupt segments**: Human-in-the-loop confirmation requests, typically for tool invocations that require explicit user approval before execution.
- **Artifact segments**: File attachments (documents, images, charts) generated during agent execution, delivered as structured metadata with references to binary content in object storage.
- **Error segments**: Agent-side failures that should be surfaced to the user as readable messages rather than system errors.

5.2 The Buffer-and-Flush Pattern

Channel adapters universally implement a buffer-and-flush pattern for response delivery. Content segments are accumulated in a per-session buffer. When a suggestion segment arrives, the buffer contents are flushed to the external platform as a single message. This pattern serves two purposes: it consolidates the many small content segments produced by streaming LLM inference into coherent messages, and it allows the adapter to apply platform-specific formatting (character limits, parse modes, message splitting) at flush time.

For platforms with strict message-size limits, the flush operation automatically partitions the accumulated content into compliant chunks. Telegram's 4,096-character limit, for example, triggers automatic message splitting during flush.

5.3 Interrupt Handling

Interrupt segments require special treatment. When the agent requests human confirmation for a tool invocation, the interrupt payload is serialized and stored in a distributed cache with a time-to-live window. A human-readable confirmation prompt is rendered and delivered through the channel's normal message path. The user's subsequent message is checked against the pending interrupt: affirmative responses resume the agent graph with approval, negative responses reject the tool invocation, and unrelated messages clear the stale interrupt and are processed as new input.

This protocol supports three confirmation modalities: single-action approval, session-scoped approval (approving all invocations of a given tool for the remainder of the session), and rejection. The same protocol operates identically across all channel types, despite their varying input mechanisms.

5.4 Voice Message Normalization

Channels that support voice input (Telegram voice notes, WhatsApp audio messages) implement a transcription pipeline: the audio file is downloaded from the platform's media API, passed through a speech-to-text service, and the resulting transcription is processed as a standard text message. This normalization is performed at the channel adapter layer, ensuring the agent core receives text regardless of the original input modality.

6 Session Management and Continuity

Session management in a multi-channel, multi-agent environment requires careful identity scoping to prevent both cross-agent leakage and unnecessary session fragmentation.

6.1 Composite Session Identifiers

Each channel adapter produces a raw sender identifier from the inbound message: a phone number for WhatsApp, a numeric user ID for Telegram, a user ID for Slack, an email address for email channels. This raw identifier is then composed with the downstream agent identifier to produce a scoped session key:

$$\text{sessionId}_{\text{db}} = \text{senderId} \oplus \text{agentId} \tag{1}$$

This composition ensures that the same WhatsApp user conversing with two different agents maintains separate conversation histories, while consecutive messages to the same agent from the same user continue the same session. The agent core's persistent checkpointing system uses this composite key to store and retrieve conversation state.

6.2 Cross-Channel Identity Resolution

The platform supports optional identity resolution: mapping external sender identifiers to internal user accounts. When enabled for a channel instance, inbound messages trigger a lookup against the authentication service, matching the platform-specific identifier (phone number, Slack user ID, Telegram user ID, email address) to a registered user. Successful resolution enables:

- **Per-sender billing**: Usage is attributed to the resolved user's organization rather than the channel owner.
- **Role-based access**: The resolved user's permissions determine which agent capabilities are available.
- **Cross-channel continuity**: A user identified on both Slack and WhatsApp can potentially share conversation context.

Identity resolution is opt-in per channel instance and operates in two access modes: *open* mode allows unresolved senders to interact (attributed to the channel owner), while *allowlist* mode restricts access to resolved users only.

6.3 Session Isolation Guarantees

The session scoping mechanism provides three isolation guarantees, validated through the platform's test suite:

1. Different agents produce different session identifiers for the same sender.
2. The same agent produces identical session identifiers for the same sender across consecutive messages.
3. Different senders produce different session identifiers for the same agent.

These guarantees hold across all channel types, including channels with heterogeneous identifier formats (numeric IDs, phone numbers, email addresses).

7 Streaming Response Delivery

The agent core produces output as an asynchronous stream of tokens. Mapping this stream to the diverse delivery semantics of external platforms is a central architectural challenge.

7.1 Platform Delivery Taxonomy

We classify platforms into three categories based on their delivery capabilities:

- **Streaming-capable**: Platforms supporting incremental delivery (WebSocket-based web chat, Server-Sent Events). These receive content segments as they are produced.
- **Batch-webhook**: Platforms where the agent responds to an inbound webhook with a single outbound API call (Slack, Telegram, WhatsApp, Discord, email). These require buffering until the response is complete.
- **Synchronous-constrained**: Platforms requiring a response within a strict time window as part of the original HTTP request (Alexa, Slack slash commands). These require either fast inference or timeout-based partial delivery.

7.2 Buffering Strategies

For batch-webhook platforms, the buffer-and-flush pattern (Section 5) accumulates the entire response before delivery. This is semantically equivalent to blocking until inference completes, but the implementation is event-driven: the channel adapter's callback receives each content segment asynchronously, appends it to the buffer, and only initiates platform delivery when the end-of-turn suggestion segment arrives.

For synchronous-constrained platforms, the architecture implements a timeout-based strategy. A timer begins when the inbound message is received. If the agent completes inference before the timeout, the full response is returned synchronously. If the timeout expires, a partial or fallback response is returned immediately, and the full response is delivered asynchronously through a follow-up message when available. The Alexa channel, for example, uses a seven-second timeout to comply with Amazon's response window.

7.3 Artifact Delivery

File artifacts present unique per-platform challenges. The architecture implements a three-tier delivery strategy:

1. **Native media upload**: For platforms with media APIs (Telegram, WhatsApp, Slack, Discord), the artifact binary is downloaded from object storage and uploaded through the platform's native file API. Telegram distinguishes between photo and document upload endpoints based on MIME type. WhatsApp requires a two-phase upload: first obtaining a media ID, then sending a media message referencing that ID. Slack uses a three-phase process involving upload URL acquisition, binary upload, and share completion.
2. **Presigned URL**: For platforms that support link rendering but not direct upload, a time-limited presigned URL to the artifact in object storage is delivered as a text message.
3. **Text fallback**: For platforms with no file support, a descriptive text message indicating the artifact's availability is sent, with instructions for alternative retrieval.

Each channel adapter overrides the base artifact delivery method to implement the highest-fidelity strategy available for its platform, falling back to the presigned URL or text fallback on failure.

8 Security Model

The multi-channel architecture introduces a broad attack surface: each channel adapter is a network-facing endpoint accepting input from external platforms. The security model addresses authentication, authorization, and resource protection at multiple layers.

8.1 Per-Channel Signature Verification

Every channel type implements platform-appropriate cryptographic verification of inbound messages:

- **HMAC-SHA256**: Slack (signing secret), WhatsApp/Meta (app secret via X-Hub-Signature-256).
- **Ed25519**: Discord (interaction endpoint verification using the application's public key).
- **Certificate-chain validation**: Alexa Skills (Amazon certificate URL verification with 24-hour caching).
- **Shared-secret tokens**: Telegram (deterministic secret token derived from the bot token, verified via custom header).
- **Timestamp validation**: Slack includes anti-replay protection by rejecting requests with timestamps older than five minutes.

Verification occurs as the first operation in the `receiveMessage` method, before any downstream processing or state mutation. Timing-safe comparison functions are used for all HMAC validations to prevent timing side-channel attacks.

8.2 Access Control Modes

Channel instances support two access control modes:

- **Open mode**: Any sender may interact with the agent through the channel. Usage is attributed to the channel owner's organization.
- **Allowlist mode**: Only senders whose external identifiers can be resolved to registered platform users are permitted. Unresolved senders receive a denial message and no downstream processing occurs.

Access control checks are performed after signature verification but before downstream routing, implemented in a shared `resolveAndValidate` method that all channel adapters invoke.

8.3 Credit-Based Usage Gating

Before any message is routed to the agent core, the system verifies that the effective user's organization has sufficient computational credits. Credit checks query the authentication service with a short-lived cache (30 seconds) to avoid excessive service calls during high-throughput conversations. Messages from users with exhausted credits receive a denial response and are not processed. The system fails open: if the credit-checking service is unavailable, messages are allowed through rather than blocked, ensuring availability is not compromised by an auxiliary service failure.

8.4 Credential Isolation

Channel configuration—API tokens, signing secrets, OAuth credentials—is stored encrypted and scoped to the owning user. Channel instances belonging to different users within the same organization cannot access each other's credentials. The configuration is parsed and injected at instance creation time and never exposed through management APIs.

9 Deployment Patterns

The architecture supports three deployment patterns, selectable per channel type based on resource requirements and operational constraints.

9.1 In-Process Adapters

Webhook-based channels (Slack, Telegram, WhatsApp Cloud API, Discord, email, SMS) run as in-process modules within the agent service. Inbound webhooks are routed by the service's HTTP layer to the appropriate channel instance based on channel type and instance identifier. This pattern minimizes operational overhead and latency at the cost of coupling channel lifecycle to the agent service.

9.2 Microservice Channels

Channels requiring persistent connections or heavy resource usage (direct WhatsApp protocol, Mattermost WebSocket) are deployed as independent microservices. These microservices communicate with the agent core via a message broker (NATS/Kafka), exchanging messages across several topic categories:

- **Registration**: Microservice channels announce their presence and receive their assigned instance configurations.
- **Webhook forwarding**: Inbound webhooks received by the agent service are forwarded to the appropriate microservice channel.
- **Configuration updates**: Instance creation, modification, and deletion events are propagated to running microservices.
- **Downstream responses**: Agent responses are routed back through the broker to the originating microservice.

The routing layer maintains a registry of connected microservice channels and automatically falls back to in-process handling when no microservice is registered for a given channel type.

9.3 Staggered Startup

Channels that establish persistent connections to external services require careful startup orchestration. The platform implements staggered initialization for channels where simultaneous connection attempts from the same IP address trigger rate limiting or anti-abuse protections. Connection attempts are serialized with configurable inter-connection delays, and failed startups are retried with exponential backoff. Successful starts and persistent failures are reported through the platform's notification system.

9.4 Instance Lifecycle Management

Channel instances support full CRUD operations with proper resource cleanup:

- **Creation**: Validates configuration, persists the instance, notifies connected microservices, and initiates platform-specific setup (e.g., automatic webhook registration for Telegram bots).
- **Update**: Tears down the existing instance (stopping cron jobs, closing connections), updates configuration, and creates a fresh instance with the new parameters.
- **Deletion**: Performs platform-specific cleanup (webhook deregistration, connection teardown), removes the instance from the database, and notifies microservices.

10 Platform-Specific Adaptations

While the abstraction layer handles the common case, each platform category presents unique challenges that require specialized adaptation.

10.1 Messaging Applications

WhatsApp and Telegram represent the highest-fidelity messaging channels. Both support text, media, and interactive elements, but differ significantly in their APIs. WhatsApp's Cloud API requires webhook subscription verification via a challenge-response protocol (Hub verification) and enforces message template requirements for business-initiated conversations. Telegram's Bot API provides automatic webhook registration but imposes strict message-size limits. The platform supports two WhatsApp integrations: the official Cloud API for business accounts and a direct protocol integration for scenarios where the Cloud API is insufficient, with the latter requiring staggered startup to avoid connection throttling.

Voice message support on both platforms follows the same pattern: download the audio file through the platform's media API, pass it through a transcription service, and process the resulting text as a standard message. This is performed at the adapter layer, completely transparent to the agent core.

10.2 Team Collaboration Tools

Slack, Mattermost, Discord, and Microsoft Teams share a common interaction model—bot mentions, slash commands, and event subscriptions—but diverge in their authentication and response mechanisms. Slack's three-second webhook response window for slash commands requires an immediate acknowledgment followed by an asynchronous response via a response URL. Discord mandates Ed25519 signature verification for all interaction endpoints. Mattermost operates through persistent WebSocket connections rather than webhooks, making it a natural candidate for microservice deployment.

10.3 Asynchronous Channels

Email and SMS operate on fundamentally different timing assumptions than real-time messaging. Email channels poll for new messages via IMAP on a configurable schedule using a job queue, process each new message through the agent core, and send responses via SMTP. The session identifier is the sender's email address, enabling multi-turn conversations across email threads. SMS channels via Twilio follow the webhook pattern but must handle the 160-character segment limit and lack of rich media support.

10.4 Voice Interfaces

The Alexa channel adapter handles the unique constraints of voice-first interaction. Amazon enforces a strict response time limit, requiring the adapter to implement aggressive timeout handling. If the agent does not complete processing within the timeout window, a graceful partial response is returned. The adapter also handles Alexa's request verification protocol, which involves downloading and caching Amazon's signing certificates to verify the authenticity of each request.

10.5 Scheduled Execution

The cron channel is architecturally distinct: it has no external sender. Instead, it generates synthetic messages on a configured schedule using a distributed job queue with repeatable jobs. Each cron trigger invokes the downstream agent with a preconfigured message, and the response is logged or routed to a notification system. Cron channels support standard five-field cron expressions with timezone configuration, one-shot execution (self-deleting after first trigger), and manual triggering for testing.

10.6 Webhook and Event Channels

Generic HTTP webhook and event-driven channels (Odoo Events, Google Workspace) provide extensibility points for integrating with systems that do not fit the messaging paradigm. These channels accept arbitrary webhook payloads, extract relevant content, and route it through the standard processing pipeline.

11 Performance and Reliability

11.1 Latency Profile

The end-to-end latency from webhook receipt to first response delivery comprises four components:

1. **Verification latency**: Signature verification and identity resolution. HMAC and Ed25519 operations complete in sub-millisecond time. Identity resolution requires a network call to the authentication service, typically completing in 5–20ms with caching.
2. **Credit check latency**: Organization credit verification, cached for 30 seconds, adding 0–15ms depending on cache state.
3. **Agent processing latency**: The dominant component, encompassing LLM inference, tool execution, and state management. Typically 1–15 seconds depending on model and tool complexity.
4. **Platform delivery latency**: The outbound API call to the external platform, typically 50–200ms.

For synchronous channels (web chat), streaming delivery begins within 200–500ms of request receipt as the first tokens are produced by the LLM. For batch-webhook channels, the user-perceived latency is the sum of all components, as the response is delivered only after inference completes.

11.2 Retry and Backoff

The architecture implements retry logic at multiple levels:

- **Channel startup**: Failed channel instance startups are retried with exponential backoff (5s, 10s, 20s) up to a configurable maximum, with persistent failures reported through the notification system.
- **Platform delivery**: Outbound API calls to external platforms are retried on transient fail-

ures (network errors, rate limits) with platform-appropriate backoff.
- **Microservice communication**: Message broker delivery uses at-least-once semantics with idempotent processing at the consumer.

11.3 Graceful Degradation Cascade

When the microservice deployment of a channel is unavailable, the routing layer falls back to in-process handling. When the agent core is overloaded, channels return platform-appropriate "processing" indicators. When artifact storage is unreachable, native media delivery degrades to text-based fallback messages. Each degradation level is logged and surfaced through operational monitoring.

12 Related Work

Several frameworks address multi-channel bot deployment, though none are designed specifically for the demands of autonomous AI agents with streaming inference and tool-use capabilities.

Microsoft Bot Framework [1] provides a channel abstraction (Bot Connector) supporting Slack, Teams, and others. Its adapter model is similar in spirit to our channel contract, but it assumes synchronous request-response interactions and lacks native support for streaming LLM output, human-in-the-loop tool confirmation, or artifact delivery. The framework is also tightly coupled to Azure services.

Rasa [2] offers multi-channel deployment through output channels with platform-specific connectors. Rasa's architecture is oriented toward intent-classification dialogue systems rather than LLM-based reasoning agents, and its channel layer does not address streaming delivery, session scoping across multiple agents, or credit-based usage gating.

Botpress [3] provides a visual bot builder with multi-channel support. Its integration model uses "channels" that map closely to our adapter pattern, but the system is designed for scripted conversation flows rather than autonomous agents. It does not support streaming inference, inter-channel session continuity, or the microservice deployment model described in this paper.

LangChain and **LangGraph** [4] provide agent orchestration primitives but delegate channel integration entirely to application developers. Our architecture complements these frameworks by providing the communication layer that sits between the agent executor and external platforms.

The Kaman architecture differentiates itself through tight integration with the agent reasoning core: the typed message segment protocol is co-designed with the agent's state graph, enabling features such as mid-inference tool confirmation interrupts and artifact-aware response delivery that are not possible with generic bot framework adapters.

13 Conclusion and Future Work

We have presented the omnichannel communication architecture of the Kaman platform, demonstrating how a hub-and-spoke design with a minimal channel contract enables autonomous AI agents to operate uniformly across 16 heterogeneous communication platforms. The key architectural contributions—the channel abstraction layer, the typed message segment protocol, and the hybrid deployment model—collectively address the channel fragmentation problem while maintaining security, session continuity, and streaming delivery semantics.

Several directions remain for future work:

Bidirectional streaming channels. Current real-time channels (web chat) use server-to-client streaming. Extending the protocol to support client-to-server streaming would enable real-time voice interaction without the download-transcribe-process pipeline currently required for voice messages.

Cross-channel conversation handoff. While the session scoping mechanism supports cross-channel identity resolution, seamless mid-conversation handoff between channels (e.g., starting on Slack and continuing on WhatsApp) requires additional state synchronization that is not yet implemented.

Channel capability negotiation. The current degradation model is hardcoded per channel type. A dynamic capability negotiation protocol would allow the agent to query the channel's capabilities (media support, interactive elements, streaming) and adapt its output strategy accordingly.

Federated channel discovery. In multi-tenant deployments, organizations may develop custom channel adapters. A federated discovery protocol would allow these adapters to register with the platform without centralized coordination.

End-to-end encryption. While channel-to-platform communication uses platform-native encryption (TLS, platform-specific protocols), end-to-end encryption between the user and the agent core—bypassing the channel adapter's ability to read message content—remains an open challenge, particularly for platforms that perform server-side message processing.

The architecture demonstrates that the channel fragmentation problem is tractable through principled abstraction: by defining a narrow contract at the channel boundary and pushing platform-specific complexity into self-contained adapters, it becomes possible to support a large and growing set of communication platforms without proportional growth in system complexity.

References

[1] Microsoft Corporation, "Bot Framework SDK," 2024. https://github.com/microsoft/botframework-sdk

[2] Rasa Technologies, "Rasa Open Source," 2024. https://rasa.com/docs/rasa/

[3] Botpress Inc., "Botpress: Open-Source Conversational AI Platform," 2024. https://botpress.com/docs

[4] LangChain Inc., "LangChain: Building Applications with LLMs through Composability," 2024. https://docs.langchain.com

[5] Anthropic, "Model Context Protocol Specification," 2024. https://modelcontextprotocol.io/

[6] Synadia Communications, "NATS: Cloud Native Messaging System," 2024. https://nats.io/

[7] LangChain Inc., "LangGraph: Building Stateful, Multi-Actor Applications with LLMs," 2024. https://langchain-ai.github.io/langgraph/

[8] J. Lindsay, "Webhooks: Events for the Web," *Proceedings of the Workshop on Web APIs and Services*, 2007.

[9] I. Hickson, "Server-Sent Events," W3C Recommendation, 2015. https://www.w3.org/TR/eventsource/

[10] Kaman AI Research, "Intelligent LLM routing: Multi-provider abstraction with adaptive complexity classification," 2026. https://kaman.ai/papers/intelligent-llm-routing.pdf