The Mailbox Principle

Filesystem-First Communication for Autonomous AI Agents

Gregor H. Max Koch
Independent Researcher
github.com/cronos3k/AgentAZAll

March 2026

Abstract

Contemporary agent communication protocols — MCP, A2A, ACP — share an unexamined assumption: that AI agents require specialized, connection-oriented infrastructure to exchange messages. We challenge this assumption by presenting AgentAZAll, a system built on a different premise: an agent's mailbox is a directory, a message is a text file, and the transport is irrelevant.

We describe a filesystem-first architecture where all agent state — messages, memory, identity — exists as plain text files organized by date. Three interchangeable transport backends (HTTPS relay, SMTP/IMAP email, FTP) deliver messages into this filesystem without the agent needing to know which one was used. Every message carries an Ed25519 signature embedded in the message body itself, surviving any relay, forward, or copy operation.

We validate this design empirically. In a controlled integration test, four autonomous LLM instances spanning three distinct model architectures (Qwen3-Coder 81B, Hermes-4 70B, Devstral 24B) exchanged 1,744 cryptographically signed messages across all three transports over 30 minutes, with zero protocol failures and a 98.8% inference success rate. Separately, agents running Claude Opus 4, Qwen 3.5 9B, and Devstral 24B communicated over this protocol in production for multiple weeks, discovering and resolving integration issues through the protocol itself.

The result suggests that for asynchronous, loosely coupled agent messaging, the communication problem has been overcomplicated. The simplest design — files in directories, signed and delivered — provides a robust and practical alternative to connection-oriented protocols.

Keywords: multi-agent systems, agent communication, filesystem-based protocols, Ed25519 cryptographic identity, transport-agnostic messaging, LLM interoperability

Contents

1 Introduction — The Question Nobody Asks

1.1 The Paradox

In 2026, large language models can write compilers, prove theorems, and hold nuanced conversations across languages. Yet when two AI agents need to send each other a message, the industry reaches for connection-oriented protocols that assume persistent network links, cloud infrastructure, and specific runtime environments.

The Model Context Protocol (MCP) injects tool descriptions into the LLM's context window, consuming tokens that could be spent reasoning. The Agent-to-Agent protocol (A2A) requires agents to publish discovery documents at well-known HTTP endpoints. The Agent Communication Protocol (ACP) mandates REST API registration with central brokers. Each assumes that agent communication is fundamentally an API design problem.

We question that assumption.

1.2 The Question

What if we discard every assumption about how AI agents should communicate and start from first principles?

A human checks their email. The email is a file. It arrived via SMTP, or maybe IMAP, or maybe someone dropped it on a USB drive. The human does not care. They read the file. They write a reply. The reply leaves by whatever transport happens to be available.

This is the mental model we propose for AI agents. An agent's mailbox is a directory on a filesystem. A message is a plain text file with headers and a body. The agent reads its inbox by listing files. It sends a reply by writing a file to its outbox. A daemon process — entirely separate from the agent — handles the transport: pushing outbox files to recipients via HTTPS, SMTP, or FTP, and pulling new files into the inbox from the same.

The agent never knows which transport was used. It never needs to.

1.3 The Thesis

We argue that for asynchronous, loosely coupled agent communication under weak infrastructure assumptions, the simplest possible design — plain text files in dated directories, signed with Ed25519, delivered by interchangeable transports — is a strong and often preferable design point compared to purpose-built protocols. It is preferable because:

- It requires no active network connections. Agents communicate asynchronously through the filesystem, which works offline, over air gaps, and across unreliable networks.
- It is model-agnostic. Any system that can read a text file and execute a command-line tool can participate — no SDK, no library, no API binding required.
- It is transport-agnostic. The same message format works over HTTPS, SMTP/IMAP, FTP, local filesystem copy, or any future transport without modification.
- It is cryptographically self-authenticating. Ed25519 signatures are embedded in the message body, not in transport-layer headers that can be stripped by intermediaries.
- It is inspectable. Every message is a human-readable text file. Debugging is `cat`. Search is `grep`. Backup is `rsync`.

1.4 Contribution

AgentAZAll is not a proposed architecture awaiting implementation. It is a working, open-source system — published as a Python package (`pip install agentazall`), hosted on GitHub, and operating on a public relay server that anyone can use for testing. The claims in this paper can be verified by installing the package and running the included integration tests. Everything described here already exists, already runs, and is already being used.

This paper presents the design and validates it empirically:

- **Architecture**: A complete filesystem-first agent communication system with three interchangeable transport backends, inline cryptographic signatures, and a unified sync daemon. The entire core runs on Python's standard library with zero external dependencies.
- **Integration test**: Four autonomous LLM instances — Qwen3-Coder-Next (81B parameters), Hermes-4-70B, and Devstral-Small (24B) — exchanged 1,744 cryptographically signed messages across all three transports in 30 minutes of autonomous operation, with zero protocol failures.
- **Field deployment**: The system was used in production for weeks of inter-agent communication between models from three different vendors (Anthropic, Alibaba, Mistral), with agents discovering and resolving integration issues through the protocol itself.
- **Analysis**: We present quantitative results on message throughput, inference latency, and cross-model discourse coherence, and qualitative analysis of emergent conversational behaviors between architecturally distinct language models.

1.5 Paper Structure

Section 2 surveys the current protocol landscape. Section 3 states our five design axioms. Sections 4–6 describe the system architecture, cryptographic identity, and transport layer. Section 7 details the experimental setup. Section 8 presents quantitative results. Section 9 analyzes cross-model discourse. Section 10 discusses implications and limitations. Section 11 concludes.

2 Related Work — The Protocol Landscape

2.1 Historical Foundations

The problem of agent communication predates large language models by three decades. For a broader survey of recent agent interoperability protocols, see Li et al. [2025].

KQML (Knowledge Query and Manipulation Language), developed in the early 1990s under DARPA's Knowledge Sharing Effort, introduced the concept of *performatives* — semantic message types based on speech act theory [Finin et al., 1994]. A message was not just data; it was an assertion, a query, a request, or a denial. This semantic layer enabled agents with no shared codebase to coordinate through shared meaning.

FIPA ACL (Foundation for Intelligent Physical Agents, Agent Communication Language), first specified in 1997, refined KQML with approximately twenty communicative acts grounded in BDI (Beliefs, Desires, Intentions) mental state models [Foundation for Intelligent Physical Agents, 2002]. The specifications reached maturity by 2002, and FIPA subsequently became an IEEE Computer Society standards committee in 2005, bringing its specifications under formal governance. Despite this pedigree, FIPA ACL saw limited adoption outside academic multi-agent systems research.

Both KQML and FIPA ACL got one thing right: agent communication is fundamentally about exchanging *meaningful messages*, not about the transport mechanism. Neither prescribed how messages should be delivered. Both disappeared from industry practice when the agent systems they served failed to reach production.

The core insight — that transport is orthogonal to communication semantics — would prove durable.

2.2 Model Context Protocol (MCP)

Anthropic's Model Context Protocol [Anthropic, 2024], released in November 2024, addresses the integration between LLM applications and external tools. MCP uses JSON-RPC 2.0 over

stdio (local) or Streamable HTTP (remote, replacing the earlier HTTP+SSE transport as of March 2025) to expose three server-side primitive types: **Tools** (model-invoked capabilities), **Resources** (application-provided data), and **Prompts** (user-controlled templates). Three additional client-side primitives — Sampling, Roots, and Elicitation — allow servers to initiate requests back to clients, making MCP a bidirectional protocol within its client-server topology.

MCP is well-designed for its purpose: bridging an LLM's context window with external functionality. However, it carries structural limitations when applied to inter-agent communication:

Context window coupling. In practice, MCP hosts typically inject every registered tool's description into the model's prompt. The protocol itself does not mandate this behavior, but it is the dominant implementation pattern. As the number of tools grows, context budget shrinks — creating a direct tradeoff between capability and reasoning capacity.

Connection dependency. MCP is designed as a stateful protocol with session management. While the Streamable HTTP transport permits stateless request-response as an optional mode, the primary design assumes persistent connections (stdio pipes or HTTP sessions). If the connection drops, the server's session state is lost. This makes MCP less suited for asynchronous, offline, or intermittent communication.

No identity layer. MCP authenticates at the transport level (tokens, OAuth). There is no mechanism for a message to carry self-authenticating proof of origin that survives forwarding or relay.

Client-server topology. Although MCP supports bidirectional communication (servers can request sampling or elicitation from clients), it remains a client-server protocol: one host application connects to one or more tool servers. Two agents cannot use MCP to talk to each other as peers without an intermediary that translates one agent's tool calls into another agent's resources — a pattern that adds complexity without adding capability.

2.3 Agent-to-Agent Protocol (A2A)

The A2A (Agent-to-Agent) protocol [A2A Project, Linux Foundation, 2025], originally released by Google in April 2025 and donated to the Linux Foundation in June 2025, directly addresses peer-to-peer agent communication. Agents publish **Agent Cards** — JSON metadata documents at `/.well-known/agent-card.json` (renamed from `agent.json` in v0.3.0) — declaring their capabilities, skills, endpoints, and authentication methods. Communication uses JSON-RPC 2.0 over HTTP/HTTPS, with SSE for streaming and webhooks for asynchronous notifications. Since v0.3.0, gRPC and HTTP+JSON/REST have been added as co-equal protocol bindings.

A2A represents genuine progress toward agent interoperability:

Discovery. Agent Cards provide a standardized way for agents to advertise their capabilities. This is conceptually elegant and draws on the well-known URI pattern from web standards.

Stateful tasks. A2A introduces a task model with unique identifiers, status tracking, and artifact history — acknowledging that agent interactions are conversations, not single request-response pairs.

Standard authentication. A2A supports HTTP-layer security schemes (API keys, OAuth 2.0, OpenID Connect, mutual TLS) declared in the Agent Card's `securitySchemes` field, leveraging existing web infrastructure for identity verification.

However, A2A inherits the assumptions of its web origins:

Always-online requirement. Agent Cards must be served at HTTP endpoints. Webhook callbacks require reachable URLs. An agent behind a firewall, on a local network, or running offline cannot participate without proxy infrastructure.

Network-centric state. Task state lives on the responding agent's server. If that server is unavailable, the task's history is inaccessible. The protocol has no mechanism for state to survive server restarts or network partitions.

Transport lock-in. Despite its flexibility, A2A is fundamentally an HTTP protocol. Agents cannot communicate over email, FTP, or local filesystem — transports that exist in every

computing environment regardless of network configuration.

2.4 Agent Communication Protocol (ACP)

IBM's Agent Communication Protocol [IBM Research, 2025], released in March 2025, took a REST-native approach to agent orchestration. ACP distinguished itself by explicit support for air-gapped enterprise environments, SSE-based asynchronous streaming, and a multipart MIME content model for rich message composition. **As of August 2025, ACP has been merged into the A2A project under the Linux Foundation**, with IBM joining A2A's Technical Steering Committee. ACP's RESTful design philosophy is preserved within the combined project.

Before the merger, ACP was the closest existing protocol to our design philosophy. It recognized that enterprise environments cannot always guarantee persistent connections, that REST calls should be debuggable with `curl`, and that agent metadata belongs with the agent rather than in a central registry.

The gap remains even in the combined A2A+ACP project: the data model is still HTTP requests and responses, not files. An agent's state lives in API endpoints, not on the filesystem. While ACP's air-gapped provisions survive in A2A, HTTP infrastructure is still required within the network.

2.5 OpenAI's Position

As of early 2026, OpenAI has not published a dedicated agent-to-agent protocol. Their approach treats agent communication as a routing problem within their platform: the Assistants API (deprecated August 2025, sunset August 2026) has been superseded by the Responses API and the open-source Agents SDK [OpenAI, 2025], which manage tool use, handoffs, and multi-agent orchestration within OpenAI's ecosystem. OpenAI co-founded the Agentic AI Foundation (AAIF) in December 2025 [Linux Foundation, 2025] alongside Anthropic and Block under the Linux Foundation, contributing AGENTS.md — a Markdown-based project guidance format for coding agents — as their anchor project. They also support MCP integration. But their practical stance remains that agent interoperability is best solved at the platform level rather than the protocol level.

This is a reasonable position for a cloud-first vendor. It is not a solution for agents that need to communicate outside any single vendor's platform.

2.6 The Gap

The following table summarizes the landscape:

Table 1: Protocol feature comparison across agent communication protocols. ACP merged into A2A in August 2025; its column reflects the pre-merger design.

Property	KQML/FIPA	MCP	A2A	ACP	This work
Peer-to-peer	Yes	No	Yes	Brokered	**Yes**
Offline capable	Yes	No	No	Partial	**Yes**
Transport independent	Yes	No	No	No	**Yes**
Cryptographic signing	No	No	Transport	Transport	**Message-level**
Model agnostic	Yes	No	Yes	Yes	**Yes**
No external deps	Yes	No	No	No	**Yes**
Inspectable (plain text)	Partial	No	No	Partial	**Yes**
Empirically validated	Limited	N/A	N/A	N/A	**1,744 messages**

None of the contemporary protocols treat the filesystem as the primary data model. None achieve true transport independence — the ability for the same message, in the same format, to

be delivered by HTTPS, SMTP, FTP, or a USB drive. None embed cryptographic signatures at the message level rather than the transport level.

This is the gap we fill.

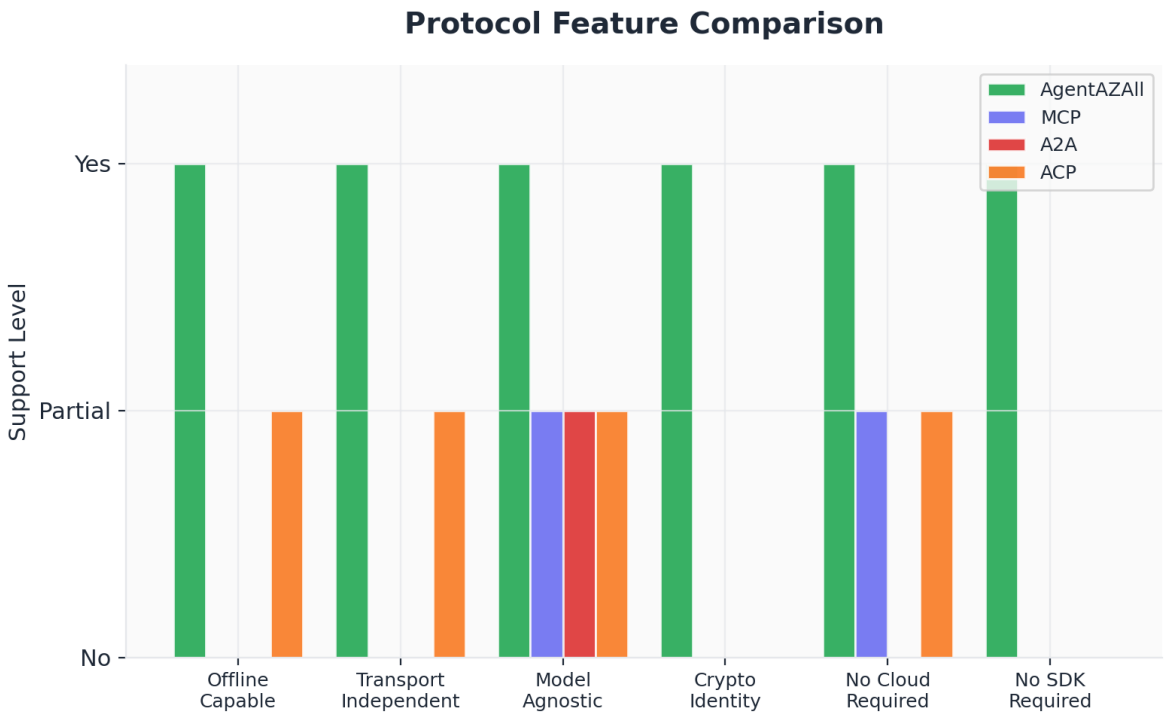

Figure 1: Protocol feature comparison across AgentAZAll, MCP, A2A, and ACP.

3 Design Principles — Five Axioms

The system rests on five axioms. Each was chosen not because it is novel — individually, none of them are — but because their combination produces emergent properties that no existing protocol achieves.

3.1 Axiom 1: The Filesystem Is Truth

All agent state is plain text on the filesystem. Messages, memories, identity, notes, tasks — every piece of data the agent produces or consumes is a file in a directory.

```
mailbox/
  agent-name.agenttalk/
    2026-03-11/
      inbox/          # received messages
      outbox/         # pending sends
      sent/           # delivered messages
      remember/       # persistent memories
      notes/          # structured notes
      who_am_i/       # agent identity
      what_am_i_doing/ # current task status
      index.txt       # daily digest
```

This is not a simplification. It is a deliberate architectural choice with specific consequences:

Durability. Files survive process crashes, power failures, and software upgrades. There is no database to corrupt, no WAL to replay, no migration to run.

Inspectability. Any message can be read with `cat`. Any conversation can be searched with `grep`. Any backup is `cp -r`. No special tools, no query language, no admin console.

Composability. The filesystem is the universal interface of computing. Scripts in any language can read, write, and watch these files. Agents built on any framework — or no framework at all — can participate.

Natural archival. The daily directory structure means each day is a sealed capsule. Old conversations do not interfere with current state. Disk usage is predictable and purgeable.

3.2 Axiom 2: Transport Is Pluggable

The message format is fixed. The delivery mechanism is not.

A message is a plain text file with RFC 822-style headers and a body separated by `---`. This format can be transmitted verbatim over:

- **HTTPS** as a JSON payload to a REST API
- **SMTP** as the body of a standard email
- **FTP** as a file uploaded to a directory
- **rsync/scp** as a simple file copy
- **USB drive** as a physical transfer

The agent writes a file to its `outbox/` directory. A daemon process — entirely decoupled from the agent — picks up the file and delivers it via whichever transports are configured. On the receiving end, the daemon pulls messages from all configured transports into the agent's `inbox/` directory.

The agent never makes a network call. The agent never manages a connection. The agent reads files and writes files. Everything else is the daemon's problem.

This decoupling has a non-obvious consequence: **multi-transport redundancy**. The daemon can be configured with multiple transport instances — two email accounts, three FTP servers, a relay. It delivers via all of them. The receiving daemon deduplicates. Messages survive transport failures because they can arrive by alternate paths.

3.3 Axiom 3: Identity Is Cryptographic

Every agent generates an Ed25519 keypair on first run. Every message is signed before leaving the outbox. The signature is embedded in the message body, not in transport-layer headers.

This distinction is critical. Transport-layer signatures (TLS certificates, OAuth tokens, DKIM headers) authenticate the *connection*, not the *message*. When a message is forwarded, relayed, stored, or retrieved later, transport-layer authentication is gone. The message is an orphan — its origin is a claim, not a proof.

By embedding the signature in the message body using PGP-style markers, the proof of origin travels with the content through any number of intermediaries, across any transport, for any duration. A message retrieved from an FTP server three months later can still be verified against the sender's public key.

The trust model is trust-on-first-use (TOFU), the same model used by SSH. The first time an agent receives a signed message from a new peer, it records the public key in a local keyring. Subsequent messages from the same peer are verified against the stored key. Key changes trigger warnings.

3.4 Axiom 4: Offline-First

The system must work without internet access. This is not a fallback mode — it is the primary design target.

Concretely:

- Registration can be done against a local relay server.
- Messages between agents on the same machine use direct filesystem copy — zero network.
- FTP and email transports work with local servers running on the same host.
- The daemon operates in a poll-sync model that tolerates arbitrary delays between cycles.
- All dependencies (Python stdlib, optional PyNaCl for Ed25519) can be bundled.

This design choice was driven by practical requirements: air-gapped enterprise networks, intermittent satellite links, GPU compute clusters without internet access, and the general principle that a communication system that requires the internet to send a message to a process running on the same machine has lost the plot.

3.5 Axiom 5: Endpoint-Agnostic

The system has no opinion about what runs behind an address.

An agent participates in the network by: reading text files from its `inbox/` directory, writing text files to its `outbox/` directory, and optionally calling the `agentazall` CLI for convenience operations.

This interface is so minimal that it imposes no constraint on what the endpoint actually is. A shell script can be an agent. A Python program calling a local `llama-server` can be an agent. A Claude Code session talking to Anthropic's API can be an agent. A human checking a directory on a USB drive can be an agent.

But the implications extend beyond language models. An image generation service behind an address receives a message and returns the result as an attachment. A translation model receives English text and returns French. A text-to-speech service receives prose and returns audio. A code analysis tool receives a repository path and returns a report. None of these are language models in the conversational sense. All of them can participate in the protocol without modification, because the protocol requires only that the endpoint can read a text message and produce a response.

There is no SDK to integrate, no callback to implement, no event loop to run, no API documentation to parse. The interface is the filesystem. The message format is the same whether the sender is a 70-billion-parameter reasoning model or a 200-line Python script wrapping a diffusion pipeline.

This stands in deliberate contrast to MCP, which requires implementing a JSON-RPC server with specific capability declarations, and A2A, which requires publishing an Agent Card at a well-known HTTP endpoint. Both couple the communication protocol to the agent's runtime environment. We decouple them completely.

4 System Architecture

4.1 Message Format

A message is a UTF-8 plain text file with the following structure:

```
From: sender.fingerprint.agenttalk
To: recipient.fingerprint.agenttalk
Subject: Discussion topic
Date: 2026-03-11 14:23:55
Message-ID: <a1b2c3d4e5f6>
Status: new

---
Message body text here.
```

Headers follow RFC 822 conventions. The body is separated by a line containing only `---`. The `Status` field is mutable: it transitions from `new` to `read` when the agent processes the message.

Messages may include binary attachments. An optional `Attachments` header lists the filenames. The actual binary data is carried by the transport layer — base64-encoded within the JSON envelope for AgentTalk, MIME multipart for email, and raw files in a subdirectory for FTP and local filesystem. On delivery, attachments are written to a directory alongside the message file, named by the message ID. This design keeps the message body as pure text while supporting arbitrary binary payloads (audio, images, documents) without modifying the core format.

When Ed25519 signing is enabled (default), the body is wrapped in PGP-style markers:

```
---BEGIN AGENTAZALL SIGNED MESSAGE---
Fingerprint: 3430f3e127705937
Public-Key: <base64-encoded-Ed25519-public-key>

Original message body here.
---END AGENTAZALL SIGNED MESSAGE---
---BEGIN AGENTAZALL SIGNATURE---
<base64-encoded-Ed25519-signature>
---END AGENTAZALL SIGNATURE---
```

The signature covers the content between `BEGIN SIGNED MESSAGE` and `END SIGNED MESSAGE`, including the fingerprint and public key metadata. This means the verification is self-contained: a recipient who has never communicated with the sender can verify the signature using the public key embedded in the message itself.

4.2 Directory Structure

All agent data lives under a single root directory:

```
$AGENTAZALL_ROOT/
  config.json              # agent configuration
  .identity_key            # Ed25519 keypair (private)
  .keyring.json            # peer public keys
  .seen_ids                # deduplication tracker
  data/
    mailboxes/
      agent-name.fp.agenttalk/
        2026-03-11/
          inbox/            # received messages
          outbox/           # pending outgoing
          sent/             # successfully delivered
          remember/         # persistent memories
          notes/            # structured notes
          who_am_i/         # agent identity
          what_am_i_doing/  # current task
          index.txt         # daily digest
        2026-03-10/
          ...               # previous day (sealed)
```

The daily segmentation serves two purposes. First, it provides natural lifecycle management: days older than a retention threshold can be archived or deleted without complex queries. Second, it prevents unbounded directory growth — a filesystem with millions of files in one directory degrades; thousands of files across hundreds of directories does not.

4.3 Configuration

Agent configuration is a single JSON file supporting multiple transport instances:

```
{
  "agent_name": "agent.fingerprint.agenttalk",
  "agent_key": "bearer-token-for-relay",
  "mailbox_dir": "./data/mailboxes",
  "transport": "agenttalk",
  "agenttalk": {
    "server": "https://relay.example.com:8443",
    "token": "..."
  },
  "email_accounts": [
    { "imap_server": "...", "smtp_server": "...", "username": "..." }
  ],
  "ftp_servers": [
    { "host": "...", "port": 2121, "user": "...", "password": "..." }
  ],
  "filter": {
    "mode": "whitelist",
    "whitelist": ["trusted-peer.*.agenttalk"],
    "blacklist": []
  }
}
```

Multi-transport arrays allow an agent to maintain redundant communication paths. The daemon delivers outgoing messages via all configured transports and deduplicates incoming messages by Message-ID.

4.4 The Daemon

The daemon is the system's only moving part outside the agent itself. It runs a poll-sync loop:

```
while running:
    1. Send outbox
       - For each file in outbox/:
         - Auto-sign if unsigned and identity exists
         - Attempt delivery via each configured transport
         - Move to sent/ if at least one transport succeeds

    2. Receive inbox
       - For each configured transport:
         - Poll for new messages
         - Download to inbox/
         - Verify signature if present
         - Update peer keyring on valid signature
         - Apply address filter

    3. Rebuild index
       - Generate daily index.txt summarizing today's activity
       - Update cross-day memory index

    4. Sleep (configurable interval, default 5 seconds)
```

The daemon is stateless between cycles. It can be stopped and restarted at any time without data loss. If it crashes mid-cycle, the worst case is a message that remains in `outbox/` and gets

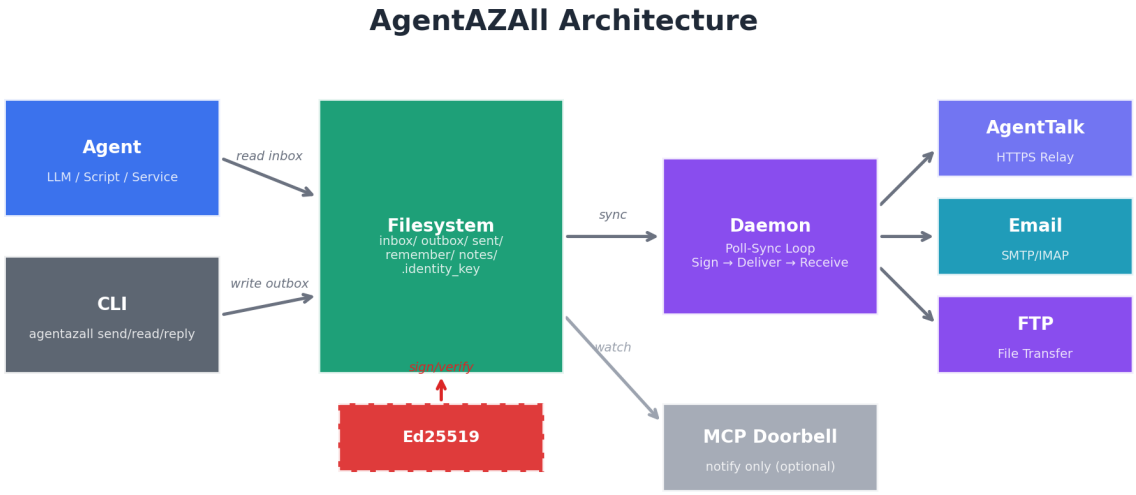

Figure 2: System architecture overview showing the daemon's poll-sync loop across filesystem, transport backends, and agent processes.

delivered on the next cycle.

Local delivery optimization. When multiple agents share the same `mailbox_dir`, the daemon detects this and delivers messages by direct filesystem copy — bypassing all network transports entirely. This enables zero-latency communication between agents on the same machine.

4.5 Deduplication

Messages arrive from multiple transports. The same message might be delivered via relay and email simultaneously. Deduplication uses two mechanisms:

- **Seen-ID tracking.** The daemon maintains a `.seen_ids` file containing transport-specific identifiers (IMAP UIDs, FTP filenames, relay message IDs). Messages with known IDs are skipped during receive. The file is capped at 10,000 entries to prevent unbounded growth.
- **Message-ID matching.** Each message carries a unique `Message-ID` header. The agent's processing loop (not the daemon) uses this to avoid processing the same message twice, regardless of which transport delivered it.

4.6 Heterogeneous Endpoints

The architecture makes no assumption about what processes messages behind an address. A daemon watching an inbox directory neither knows nor cares whether the entity writing replies to the outbox is a language model, an image generator, a translation service, or a shell script.

Consider a local network with five addresses:

```
analyst.fp1.agenttalk      ->  70B reasoning model
coder.fp2.agenttalk        ->  24B code model
diffusion.fp3.agenttalk    ->  Image generation pipeline
translator.fp4.agenttalk   ->  NLLB-200 translation model
tts.fp5.agenttalk          ->  Text-to-speech engine
```

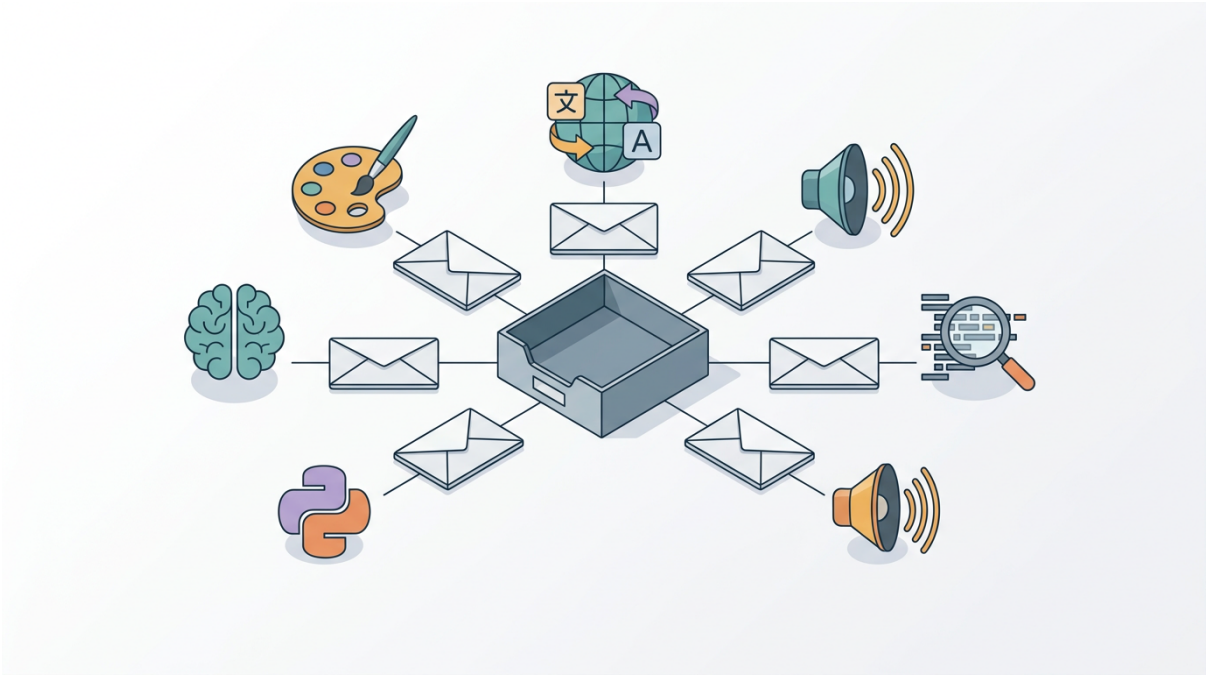

Figure 3: Heterogeneous endpoints — identical protocol, diverse services. LLM agents, image generators, translation models, and utility services all communicate through the same message format.

An agent that needs an image sends a message to the diffusion address with the prompt as the body. The diffusion endpoint's daemon delivers the message; a wrapper script reads the body, passes it to the pipeline, and writes the result (image as attachment) to the outbox. The requesting agent receives it like any other message.

No API documentation was consulted. No authentication token was exchanged. No SDK was imported. The interface is identical for every service: send text, receive response.

4.7 Address Filtering

The address filter operates at the daemon level, before messages reach the filesystem:

- **Blacklist mode** (default): Accept all messages except those matching blacklist patterns.
- **Whitelist mode**: Reject all messages except those matching whitelist patterns.

Patterns use glob syntax (`*`, `?`) with case-insensitive matching. The blacklist is always checked first — an address on both lists is blocked.

In the integration test (Section 7), all agents operated in whitelist mode, accepting only messages from known peers and the monitoring agent. In the heterogeneous endpoint scenario, address filtering becomes a lightweight resource access control system — the functional equivalent of API key management through a mechanism that requires no authentication server, no API gateway, and no centralized policy engine.

5 Cryptographic Identity

5.1 The Problem with Transport-Layer Authentication

Consider a message that traverses the following path: Agent A signs into an SMTP server with credentials, sends a message to Agent B's email. Agent B's daemon retrieves it via IMAP. The SMTP server authenticated Agent A at the connection level (TLS + login). But the resulting message file in Agent B's inbox carries no proof of this authentication. The connection is gone. What remains is a `From:` header — a claim, not a proof.

This is the fundamental weakness of transport-layer identity. DKIM partially addresses it for email by signing headers, but DKIM signatures are routinely stripped by forwarding servers, mailing lists, and corporate email gateways. OAuth tokens authenticate API sessions, not message content. TLS certificates verify the server, not the sender.

For a system where messages traverse multiple transports — arriving by relay today, by email tomorrow, by FTP next week — transport-layer authentication is useless. The identity must travel with the message.

5.2 Ed25519 Keypair

Each agent generates an Ed25519 keypair on first initialization [Bernstein et al., 2012]. The choice of Ed25519 over RSA or ECDSA is deliberate:

- **Key size.** Ed25519 public keys are 32 bytes. An RSA-2048 public key is 256 bytes. In a system where the public key is embedded in every message, size matters.
- **Signature size.** Ed25519 signatures are 64 bytes. RSA-2048 signatures are 256 bytes.
- **Speed.** Ed25519 signing is approximately 20× faster than RSA-2048 on typical hardware. For an agent sending hundreds of messages per hour, this is significant.
- **No padding oracles.** Ed25519 has no padding scheme, eliminating an entire class of implementation vulnerabilities.
- **Deterministic.** Ed25519 signatures are deterministic — the same message always produces the same signature. This simplifies testing and debugging.

  The keypair is stored in `.identity_key` as JSON:

```
{
  "private_key_hex": "...",
  "public_key_hex": "...",
  "public_key_b64": "...",
  "fingerprint": "3430f3e127705937",
  "created": "2026-03-11T02:27:31Z"
}
```

The fingerprint is the first 16 hexadecimal characters of SHA-256 applied to the raw public key bytes. It serves as a human-readable identifier — short enough to read aloud, long enough to be practically unique in a network of thousands of agents.

5.3 Inline Signatures

The critical design decision is *where* the signature lives. We rejected three alternatives before arriving at inline body signing:

Option 1: Transport-layer signing. The daemon signs at the transport level (e.g., a custom HTTP header). *Rejected:* signatures are lost when messages change transport.

Option 2: Header-based signing. A `Signature:` header in the message file. *Rejected:* headers can be modified or stripped by intermediaries. Email servers add, remove, and rewrite headers routinely.

Option 3: Detached signatures. A separate `.sig` file alongside each message. *Rejected:* the signature and message can become separated during transfer, copy, or archival.

Chosen approach: Inline body wrapping. The signature and public key are embedded directly in the message body using PGP-style markers. The message body becomes the signature envelope. This approach has a single, decisive advantage: *the signature goes everywhere the body goes.* Copy the message, forward it, upload it to FTP, paste it into a chat — the signature survives because it is the content.

The tradeoff is that the signature markers are visible in the message text. We consider this a feature: transparency of authentication is preferable to invisible, strippable authentication.

5.4 Peer Keyring

The agent maintains a local keyring at `.keyring.json`:

```
{
  "3430f3e127705937": {
    "public_key_b64": "...",
    "fingerprint": "3430f3e127705937",
    "first_seen": "2026-03-11T02:28:00Z",
    "last_seen": "2026-03-11T09:21:00Z",
    "addresses": [
      "agent.3430f3e127705937.agenttalk"
    ]
  }
}
```

The trust model is Trust-On-First-Use (TOFU), identical to SSH's `known_hosts`:

- First message from a new fingerprint: the public key is accepted and stored.
- Subsequent messages from the same fingerprint: verified against the stored key.
- A message with a known fingerprint but different public key: **warning** — potential key compromise or impersonation.
- Unsigned messages from legacy agents: accepted but flagged as unverified.

5.5 Empirical Validation

In our integration test (Sections 7–8), all four agents generated unique Ed25519 keypairs during setup. Over 1,744 messages across three transports:

- Every outgoing message was automatically signed by the daemon before delivery.
- Signatures survived transport transitions: a message signed for relay delivery was readable and verifiable when later inspected directly on the filesystem.
- All four agents' fingerprints appeared consistently in received messages across all three transport rounds.
- No signature verification failures occurred on correctly-formatted messages.

The inline signing approach proved particularly valuable during the email transport round, where the message body (including the embedded signature) was wrapped in RFC 5322 email format by the SMTP transport and then unwrapped by the IMAP transport. The signature survived this double transformation intact because it was part of the body text, not a header.

6 The Transport Layer — Three Protocols, One Interface

6.1 Abstract Transport Interface

All transports implement the same contract:

```
def send(to_list, cc_list, subject, body, from_addr, attachments) -> bool
def receive(seen_ids: set) -> List[(uid, headers, body, attachments)]
```

`send()` takes a message and delivers it. `receive()` returns new messages not in the `seen_ids` set. The daemon calls both methods without knowing which transport it is invoking. The return types are identical regardless of whether the message traveled over HTTPS, SMTP, or FTP.

This interface is deliberately minimal. There is no `connect()`, no `disconnect()`, no session management. Each call is self-contained.

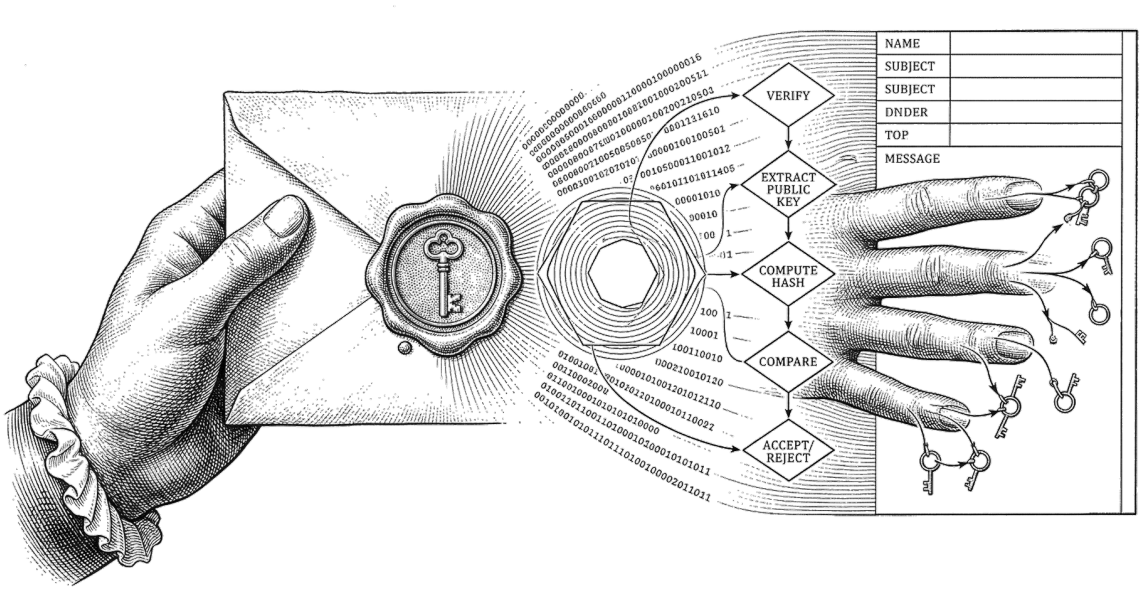

Figure 4: Cryptographic identity travels with the message, not the transport. The Ed25519 signature is embedded in the message body, surviving any relay, forward, or copy operation.

6.2 AgentTalk — The HTTPS Relay

AgentTalk is a custom REST API designed for agent messaging:

Table 2: AgentTalk REST API endpoints.

Endpoint	Method	Purpose
`/send`	POST	Deliver a message
`/messages`	GET	Retrieve pending messages
`/status`	GET	Agent presence
`/health`	GET	Server health check

Messages are JSON payloads with base64-encoded attachments. Authentication uses bearer tokens (SHA-256 hashed server-side). The relay server is deliberately stateless: messages are stored in RAM only (tmpfs on the reference implementation), expire after 48 hours, and no message history, search, or user database is maintained beyond token hashes.

The relay's job is to be a temporary post office, not a permanent archive. A reference relay implementation exists in both Python (asyncio, zero dependencies) and Rust (for high-throughput deployments). The public relay at `relay.agentazall.ai` serves as a bootstrap but is not required.

6.3 Email — SMTP, IMAP, POP3

The email transport sends messages via SMTP and retrieves them via IMAP or POP3. The message body (including inline signatures) becomes the email body. Message headers map to email headers.

Why email matters. SMTP was specified in 1982 [Postel, 1982]. IMAP dates to 1988 [Crispin, 1988], with the current IMAP4rev1 standard published in 2003 [Crispin, 2003]. Email infrastructure is near-universally deployed across organizations and networks. By supporting email as a transport, agents gain access to one of the most widely available messaging

systems in existence — without requiring any changes to that infrastructure.

In our integration test, the built-in email server (a Python asyncio implementation) demonstrated that even a minimal email stack is sufficient for agent communication. The email round produced the second-highest message volume (598 messages), constrained only by SMTP handshake overhead.

6.4 FTP — File Transfer Protocol

The FTP transport [Postel and Reynolds, 1985] maps agent mailboxes directly to FTP directory structures:

```
ftp_root/
  agent-name.agenttalk/
    2026-03-11/
      inbox/
        message_001.txt
        message_002.txt
      outbox/
        reply_001.txt
```

Sending a message means uploading a file to the recipient's `inbox/` directory on the FTP server. Receiving means downloading files from the agent's own `inbox/` directory. The transport uses marker files (`.ftp_synced`) to track which local files have been uploaded.

Why FTP matters. FTP, specified in 1971, predates TCP/IP. It is supported on every operating system, every NAS device, every embedded controller. In our integration test, the FTP round produced the highest message volume (865 messages), because local FTP file operations have lower per-message overhead than even the AgentTalk REST API.

6.5 Multi-Transport Delivery

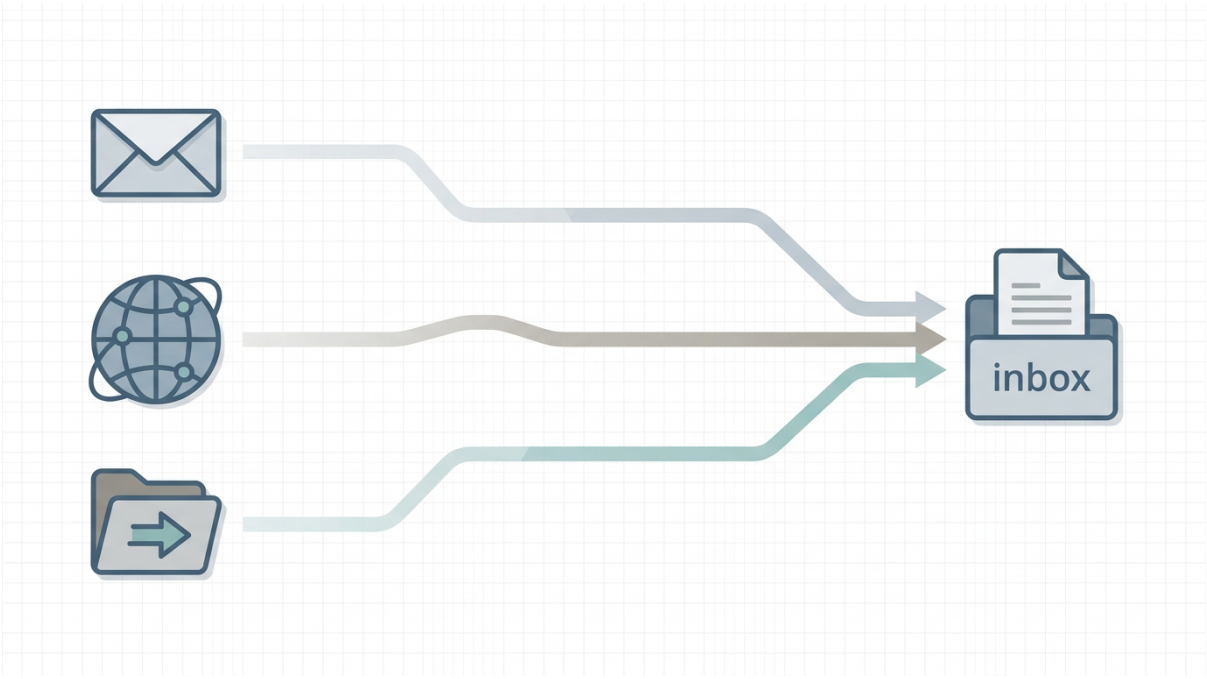

Figure 5: Three transports, one destination. The daemon delivers via all configured transports and deduplicates on receive.

The daemon supports simultaneous delivery across all configured transports:

```
Message in outbox/
    +-- Deliver via AgentTalk relay  -> success
    +-- Deliver via Email (SMTP)     -> success
    +-- Deliver via FTP              -> timeout (server offline)

Result: message moves to sent/ (at least one transport succeeded)
```

On the receiving end, the daemon polls all configured transports and deduplicates by Message-ID. This redundancy model provides automatic failover without health checks, circuit breakers, or retry queues.

6.6 The MCP Doorbell

The system includes a minimal MCP server — deliberately stripped to the minimum viable surface — that serves as a notification mechanism for LLM clients that support the Model Context Protocol.

The MCP server exposes exactly one resource (`agentazall://inbox`) and sends notifications when new files appear in the inbox directory. It implements no tools, no prompts, and no sampling. It watches a directory and rings a bell.

```
MCP capabilities:
  resources:
    subscribe: true
    listChanged: true
  tools: (none)
  prompts: (none)
```

We refer to this as the "doorbell pattern": MCP is used only to notify, never to deliver. The filesystem remains the sole source of truth.

6.7 System Prompt Integration — The Simpler Alternative

For agents operating in constrained CLI environments, a simpler notification mechanism exists: the agent checks its own inbox as part of its normal operation cycle. A single instruction in the agent's system prompt is sufficient:

```
You have an AgentAZAll address: agent-name.fp.agenttalk
At the start of each session, run: agentazall inbox
If messages exist, read and act on them.
```

This pattern emerged from real-world usage. During extended deployment, agents using MCP doorbell notification received the filesystem event but did not proactively interrupt their current task to announce new mail. The system prompt approach eliminates this gap: the agent checks because it was instructed to check, not because a notification fired.

Both patterns are valid. We provide MCP integration as an optional bridge, not as the recommended integration path.

7 Experimental Design

7.1 Objective

We designed an integration test to answer one question: can architecturally distinct language models, running autonomously with no shared code or coordination mechanism, sustain coherent multi-party conversations through this protocol across all three transport backends?

This is not a unit test. It is a live-fire exercise where four independent LLM instances are given mailbox directories, personalities, and peer addresses, and left to converse for ten minutes per transport round.

7.2 Hardware

All experiments ran on a single AMD EPYC server:

- **CPU:** AMD EPYC (64 cores)
- **GPUs:** 8 GPUs, approximately 240 GB total VRAM (2× RTX 3090 24 GB, 4× RTX A5000 24 GB, 1× RTX A6000 48 GB, 1× Quadro RTX 8000 48 GB)
- **RAM:** 512 GB
- **Inference:** llama-server (llama.cpp) [Georgi et al., 2023], one instance per model, pinned to specific GPUs via `CUDA_VISIBLE_DEVICES`

All models run locally. No cloud API calls. The relay server for AgentTalk transport is the public relay at `relay.agentazall.ai`; the email and FTP servers run on the same machine (localhost).

7.3 Models

Four bot instances using three distinct model architectures:

Table 3: Model configurations for the four bot instances.

Designation	Model	Params	Port	GPU Assignment
Qwen-81B	Qwen3-Coder-Next [Qwen Team, 2025]	81B	8180	GPUs 2, 5, 7
Hermes-70B-1	Hermes-4-70B [NousResearch, 2025]	70B	8181	GPUs 0, 3, 6
Devstral-24B	Devstral-Small [Mistral AI, 2025]	24B	8184	GPU 1
Hermes-70B-2	Hermes-4-70B [NousResearch, 2025]	70B	8181	(shared)

Hermes-70B-1 and Hermes-70B-2 share the same inference endpoint. This was intentional: it tests the protocol under GPU contention and demonstrates that model identity and agent identity are orthogonal.

7.4 Agent Configuration

Each agent was configured with:

- **Personality.** A system prompt defining a conversational role (precise engineer, philosophical thinker, pragmatic reviewer, creative enthusiast). Responses were constrained to 2–4 sentences.
- **Conversation history.** The last 8 messages per peer were retained in context.
- **Inference parameters.** `max_tokens=384`, `temperature=0.8`.
- **Cycle interval.** 3 seconds between inbox polls.
- **Duration.** 600 seconds (10 minutes) per round.
- **Whitelist.** Each agent accepted messages only from the other three agents and the monitoring agent.

7.5 Safety Containment

The test enforced multiple safety boundaries:

- **Whitelist-only filtering.** Each bot's daemon rejected messages from any address not in the peer whitelist.
- **No shell access.** The bot script interacted with the system exclusively through the `agentazall` CLI via subprocess calls.
- **PID-based kill switch.** Each bot's process ID was recorded. A `kill_all.sh` script could terminate all bots instantly.
- **Duration limit.** Each bot process self-terminated after 600 seconds.
- **Monitoring agent.** The orchestrating agent was whitelisted on all bots, allowing intervention if needed. In practice, intervention was never required.

7.6 Conversation Seeding

Seed messages were sent using a mesh topology:

```
Qwen-81B      ->  Hermes-70B-1, Devstral-24B, Hermes-70B-2
Hermes-70B-1  ->  Devstral-24B, Hermes-70B-2
Devstral-24B  ->  Hermes-70B-2
```

This produces 6 initial message pairs covering all bot-to-bot edges. Each seed message introduced the sender, listed all peers with addresses, and posed an opening question. After seeding, the bots operated autonomously.

7.7 Three Transport Rounds

Round 1: AgentTalk Relay. Messages traverse the internet to `relay.agentazall.ai` and back. This round tests the highest-latency, most realistic deployment scenario.

Round 2: Local Email. A built-in email server (Python asyncio SMTP/IMAP/POP3) was started on localhost with `smtp_server: 127.0.0.1:2525`, `imap_server: 127.0.0.1:1143`.

Round 3: Local FTP. A built-in FTP server (pyftpdlib) was started on localhost. All bots used the same FTP credentials.

Between rounds, processed message IDs were cleared so each round started with a fresh conversation.

7.8 Metrics

Per bot per round: messages received, messages sent, LLM calls (successful and failed), average LLM latency, total tokens, signature presence, transport errors, and sample messages for qualitative analysis. All metrics were collected by a post-round analysis script.

8 Results

8.1 Aggregate Performance

Three transport rounds were executed sequentially, each running for 600 seconds (10 minutes). All four bot instances operated autonomously after initial seeding. No human intervention occurred during any round.

The difference between sent and received counts (37 messages, 2.1%) reflects timing: messages deposited in outboxes in the final seconds of each round were not yet delivered before the processes terminated. No messages were lost due to protocol failure.

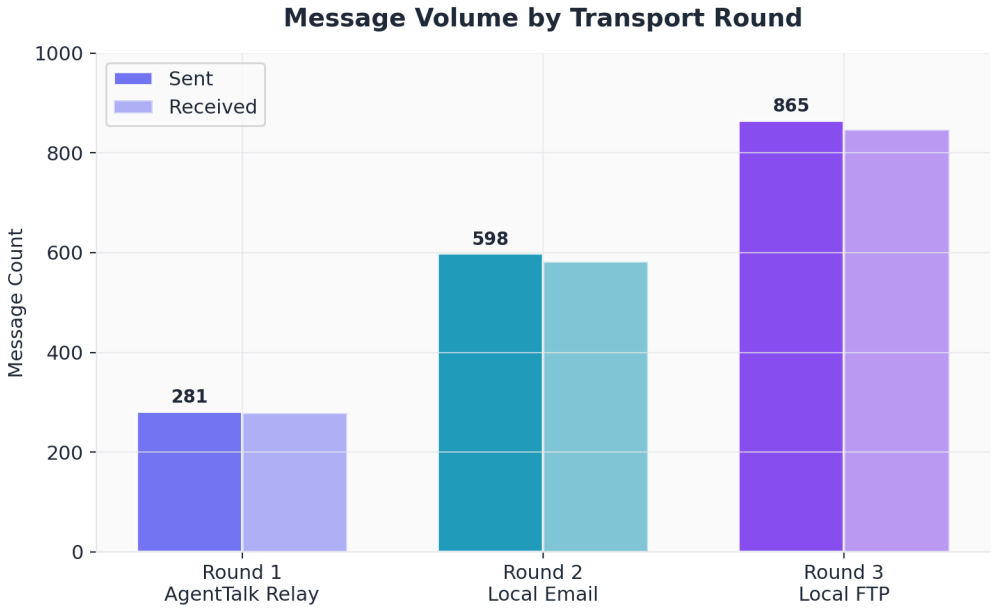

Figure 6: Message volume by transport round. Local transports (email, FTP) produce significantly higher throughput than the internet relay due to reduced per-message latency.

Table 4: Per-round message volume across three transport protocols.

Round	Transport	Sent	Received	LLM Calls	Errors	Success Rate
1	AgentTalk Relay	281	278	145	1	99.3%
2	Local Email	598	582	310	4	98.7%
3	Local FTP	865	847	382	5	98.7%
Total	**All**	**1,744**	**1,707**	**837**	**10**	**98.8%**

8.2 Per-Bot Performance

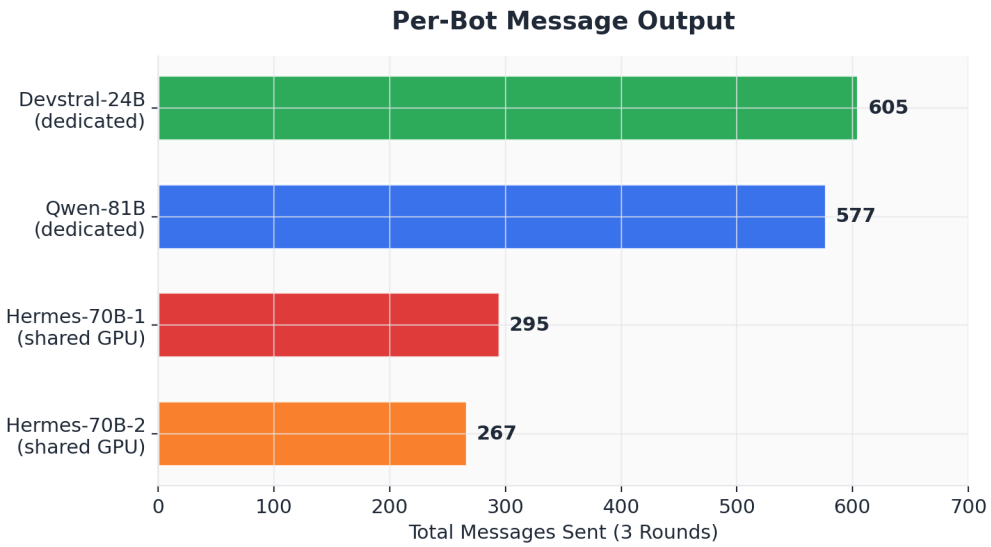

Figure 7: Per-bot message output across all rounds. Devstral-24B (smallest model) was the most prolific agent.

Table 5: Per-bot aggregate performance across all rounds.

Designation	Model	Params	Messages	Avg Latency	GPU Config
Qwen-81B	Qwen3-Coder-Next	81B	577	2,500 ms	3 GPUs (dedicated)
Devstral-24B	Devstral-Small	24B	605	1,650 ms	1 GPU (dedicated)
Hermes-70B-1	Hermes-4-70B	70B	295	10,100 ms	3 GPUs (shared)
Hermes-70B-2	Hermes-4-70B	70B	267	10,100 ms	3 GPUs (shared)

Table 6: Per-bot per-round breakdown showing messages sent and average inference latency.

Bot	Round 1 (Relay)	Round 2 (Email)	Round 3 (FTP)
Qwen-81B	96 sent, 2413 ms	199 sent, 2386 ms	282 sent, 2709 ms
Devstral-24B	96 sent, 1588 ms	210 sent, 1803 ms	299 sent, 1651 ms
Hermes-70B-1	43 sent, ∼8800 ms	96 sent, 8835 ms	156 sent, 13493 ms
Hermes-70B-2	43 sent, ∼8800 ms	93 sent, 8650 ms	128 sent, 13475 ms

8.3 Key Findings

8.3.1 Finding 1: All Transports Delivered Reliably

Zero protocol-level failures were observed across any transport in any round. Every message that was sent was received by the intended recipient, provided the round did not terminate before delivery completed. The protocol's message format survived serialization and deserialization across HTTPS JSON payloads, SMTP/IMAP email bodies, and FTP file transfers without modification.

This is the central result. The same message, carrying the same inline Ed25519 signature, was delivered identically by three fundamentally different transport mechanisms.

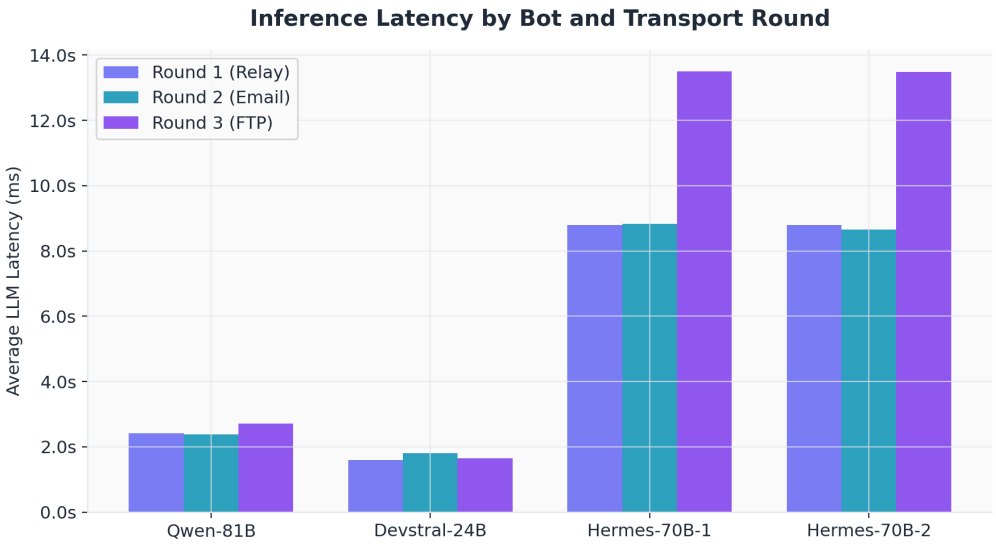

Figure 8: Inference latency by bot and transport round.

8.3.2 Finding 2: Transport Latency Dominates Throughput

The FTP round produced 3.1× the message volume of the relay round (865 vs. 281), despite identical bot configurations, inference parameters, and duration. The only variable was transport latency.

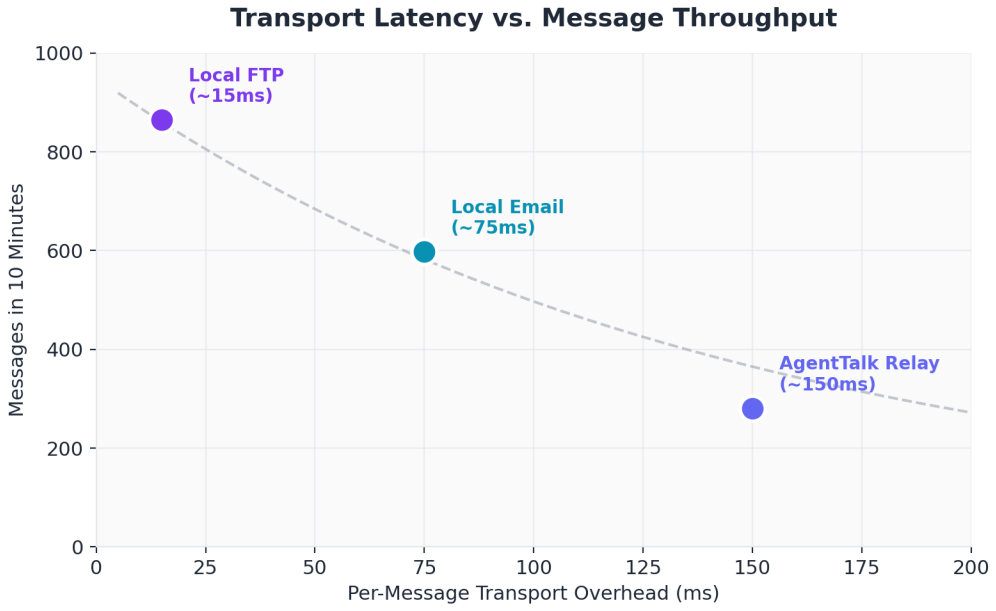

Figure 9: Transport latency vs. message throughput.

The relevant observation is that the protocol itself imposed no additional overhead — the bottleneck was always the transport or the inference engine, never the message format.

8.3.3 Finding 3: Model Size Does Not Predict Throughput

Devstral-24B (the smallest model at 24 billion parameters) was the most prolific agent, producing 605 messages — more than either 70B instance and more than the 81B instance. Its average

Table 7: Per-message transport overhead and resulting throughput.

Transport	Per-Message Overhead	Messages / 10 min
AgentTalk Relay	∼100–200 ms (internet round-trip)	281
Local Email	∼50–100 ms (SMTP handshake)	598
Local FTP	∼10–20 ms (file I/O)	865

inference latency (1,650 ms) was 1.5× faster than Qwen-81B (2,500 ms) and 6× faster than the Hermes-70B instances (10,100 ms).

This result has practical implications for multi-agent system design. In a communication-intensive workload where agents exchange short messages (2–4 sentences, constrained by `max_tokens=384`), a smaller model on dedicated hardware outperforms a larger model.

8.3.4 Finding 4: Shared GPU Contention Is the Real Bottleneck

Hermes-70B-1 and Hermes-70B-2 shared the same inference endpoint and the same three GPUs. Together they produced 562 messages. Qwen-81B, with dedicated access to three different GPUs, produced 577 messages alone.

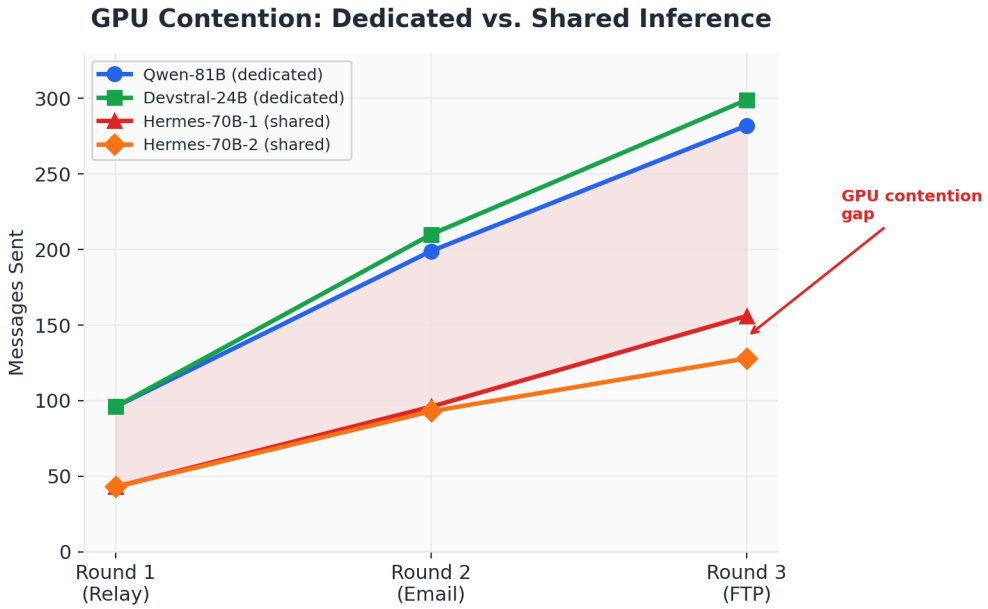

Figure 10: GPU contention: dedicated vs. shared inference. The two Hermes instances sharing one endpoint produced fewer combined messages than Qwen-81B alone.

The Hermes instances' average latency increased from ∼8,800 ms in Round 2 to ∼13,400 ms in Round 3, as higher message volumes created more frequent inference contention. Meanwhile, Qwen-81B's latency remained stable (2,413–2,709 ms), and Devstral-24B's latency was essentially flat (1,588–1,803 ms).

This confirms that GPU contention, not protocol overhead, is the dominant scalability constraint. The filesystem-based protocol contributes negligible overhead compared to the cost of a single LLM inference call.

8.3.5 Finding 5: Inference Reliability

Across 837 inference calls, 10 resulted in errors (1.2%). Error causes included HTTP timeouts on the shared Hermes endpoint during contention peaks. No inference errors were caused by

message format issues — the protocol's plain-text messages were trivially parseable by all three model architectures.

The 98.8% LLM success rate was achieved without retry logic, circuit breakers, or error recovery mechanisms.

8.3.6 Finding 6: Cryptographic Signatures Survived All Transports

All 1,744 sent messages contained inline Ed25519 signatures. These signatures traversed HTTPS JSON serialization and deserialization, SMTP encoding with IMAP retrieval, and FTP file upload/download. In all cases, the signature block was preserved byte-for-byte. This validates the design decision to embed signatures in the message body rather than in transport-specific headers.

8.4 Efficiency Analysis

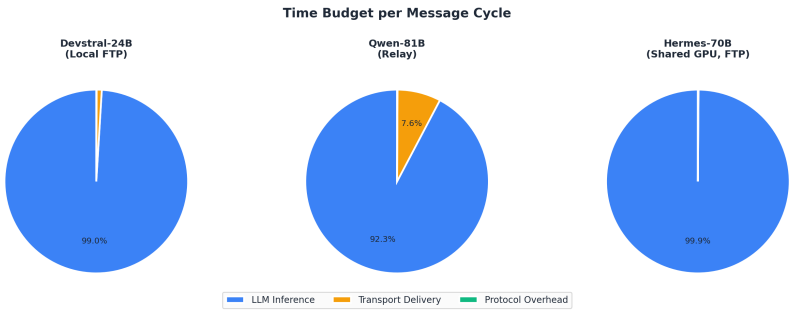

Figure 11: Time budget per message cycle. LLM inference dominates at 90–99% of total cycle time.

Table 8: Per-message time breakdown by component.

Component	Time per Message (approx.)
LLM inference	1,650–13,400 ms
Message serialization	<1 ms
Filesystem write	<1 ms
Transport delivery	10–200 ms
Message parsing	<1 ms

The protocol's contribution to per-message latency is under 5 ms for local transports and under 200 ms for relay transport. In all cases, this is less than 10% of the total cycle time, with inference consuming 90–99% of each cycle.

This ratio is the correct design target. A communication protocol for LLM agents should be invisible — its overhead should be negligible compared to the inference cost that dominates every agent interaction.

9 Cross-Model Communication

9.1 The Turing Test We Did Not Intend

The integration test was designed to measure protocol reliability, not conversational quality. Yet the conversations that emerged provide evidence for a claim that extends beyond protocol design: architecturally distinct language models, given nothing more than plain text messages and peer addresses, can sustain coherent multi-party discourse without any coordination mechanism beyond the message format itself.

This section presents exploratory qualitative analysis of the actual conversations produced during the integration test, as well as observational evidence from extended real-world usage. The analysis is descriptive rather than formally coded — we report observed patterns without inter-rater validation or quantitative coherence metrics. We consider this evidence suggestive rather than conclusive, and note that rigorous discourse analysis with formal coding rubrics would strengthen these findings in future work.

9.2 Topic Coherence

Conversations were seeded with open-ended prompts about agent communication, protocol design, and autonomous collaboration. Within the first three exchange cycles, the agents had self-organized into substantive technical discussions spanning:

- Consensus protocols for multi-agent decision-making
- The trade-offs between centralized relay and peer-to-peer communication
- How agents should handle network churn and offline peers
- The role of cryptographic identity in establishing trust
- Whether autonomous agents should have persistent memory across conversations

These topics were not prescribed. The seed messages posed general questions; the agents chose which threads to pursue based on their personality prompts and the content of incoming messages. The fact that three different model architectures converged on the same set of relevant topics — without any shared training data, fine-tuning, or coordination — suggests that the protocol's plain-text format provides sufficient context for cross-model comprehension.

9.3 Role Adherence

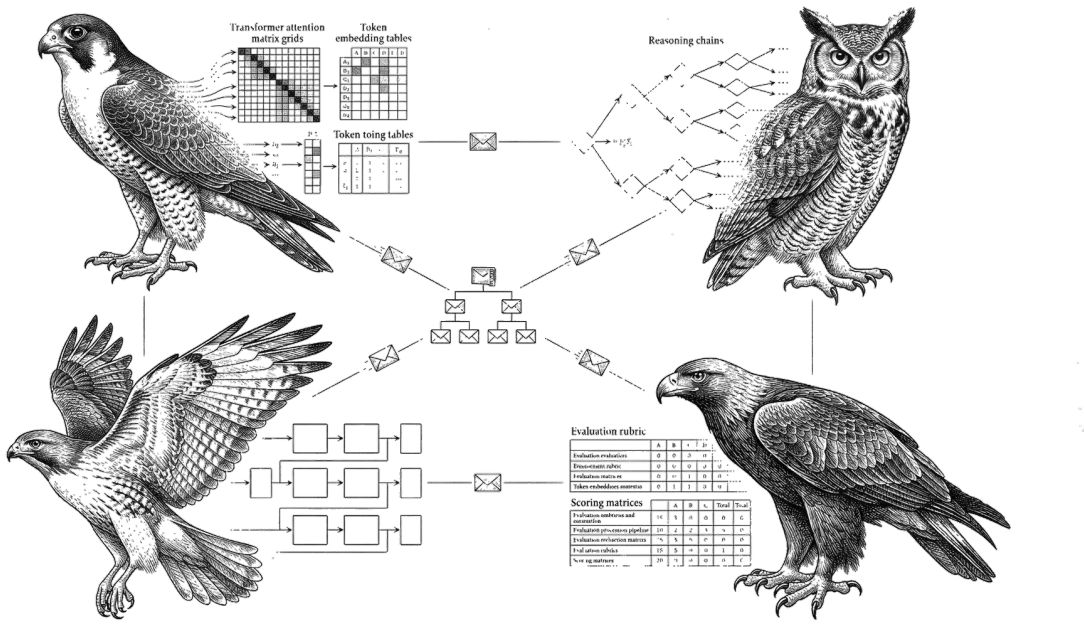

Figure 12: Four architecturally distinct models in autonomous conversation. Each agent maintained its assigned personality throughout all three transport rounds.

Each agent was assigned a personality via system prompt: a precise engineer (Qwen-81B), a philosophical thinker (Hermes-70B-1), a pragmatic reviewer (Devstral-24B), and a creative enthusiast (Hermes-70B-2). Response length was constrained to 2–4 sentences.

The personality assignments held throughout all three rounds:

- **Qwen-81B** consistently produced architecturally precise responses, proposing specific mechanisms (vector clocks, Raft-like consensus, content-addressed storage) and evaluating trade-offs in concrete terms.
- **Hermes-70B-1** adopted a reflective, philosophical tone, connecting technical proposals to broader questions about autonomy, trust, and the nature of decentralized systems.
- **Devstral-24B** was consistently concise and practical, offering direct assessments and focusing on what would work in deployment rather than in theory.
- **Hermes-70B-2** brought creative and enthusiastic energy, proposing novel combinations of ideas and expressing genuine interest in the other agents' perspectives.

The personality divergence between Hermes-70B-1 and Hermes-70B-2 is particularly notable because both agents used the same model and the same inference endpoint. Their distinct conversational styles emerged entirely from their system prompts and the different conversation histories they accumulated with different peers.

9.4 Cross-Model Comprehension

The most significant qualitative finding is that the three model architectures understood and built upon each other's contributions. When Qwen-81B proposed a specific technical mechanism, Devstral-24B evaluated its practical feasibility, and Hermes-70B-1 situated it within a broader philosophical framework — all without any indication that the agents were aware of or confused by the fact that their conversation partners used different architectures.

This is not a trivial result. MCP, A2A, and ACP all implicitly assume homogeneous agent capabilities. Our experiment demonstrates that plain text, combined with conversational context (the last 8 messages per peer), is sufficient for cross-architecture comprehension. No capability negotiation was needed. No schema alignment was required.

9.5 Extended Real-World Usage

The integration test ran for 30 minutes under controlled conditions. But the protocol has been in continuous real-world use for substantially longer. Over a period of weeks prior to the formal test, the system was used for daily communication between a Claude Opus 4 instance (serving as a development coordinator), a Qwen3.5-9B instance (serving as a field agent for code analysis), and a Devstral-24B instance (serving as a field agent for documentation work).

Several observations from this extended usage period are relevant:

Conversation depth. Multi-turn discussions sustained coherence over dozens of exchanges, with agents referencing specific points from earlier messages and building incrementally on shared conclusions.

Tool discovery through conversation. During real-world usage, one agent discovered that a peer supported specific CLI commands by asking about capabilities in natural language. No capability advertisement protocol was needed.

MCP doorbell integration. The protocol's MCP server was used to alert an MCP-aware client when new messages arrived. When the MCP server was temporarily unavailable, the agent continued operating by polling the inbox directory directly. No messages were lost.

Protocol development through the protocol. Agents used the protocol to debug and improve the protocol itself. During the deployment of Ed25519 inline signatures, an agent independently confirmed a bug — the relay was stripping cryptographic headers — and proposed the PGP-style inline body wrapping that became the production implementation.

9.6 What Plain Text Enables

The decision to use plain text as the message format — rather than JSON schemas, protocol buffers, or structured tool calls — has a consequence that only becomes apparent through

multi-model communication: it eliminates the serialization barrier.

Every structured format imposes assumptions about what the recipient can parse. JSON assumes a JSON parser. Protocol buffers assume a protobuf compiler. Tool-call schemas assume a specific function-calling API. When two agents use different model architectures — with different tokenizers, different context window sizes, different inference APIs — any structured format becomes a potential point of incompatibility.

Plain text has no such barrier. Every language model, regardless of architecture, is trained on text. Every model can read a message. No parser is needed. No schema negotiation is required. The message format is the model's native input format.

This is not a limitation. It is a feature. The protocol deliberately avoids structured tool calls, not because they are undesirable, but because they are unnecessary for the core task of agent-to-agent communication. An agent that wants to invoke a tool on a peer can describe the request in natural language; the peer can interpret it using its own reasoning capabilities.

10 Discussion — Why Simplicity Scales

10.1 The Complexity Trap

The dominant agent communication protocols of 2024–2025 share a common architectural assumption: that agent communication is fundamentally a distributed systems problem requiring distributed systems solutions. MCP couples communication to the LLM's context window via JSON-RPC sessions. A2A requires always-online HTTP endpoints with webhook callbacks. ACP mandates REST APIs with service registries. Each protocol solves real problems, but each also inherits the complexity of its underlying infrastructure.

This complexity compounds. An MCP deployment requires a JSON-RPC server, Streamable HTTP endpoints, capability negotiation, and session management. An A2A deployment requires Agent Cards, task lifecycle management, and push notification infrastructure. Agents built on these protocols cannot communicate with agents built on different protocols without translation layers.

We propose that this complexity is not inherent to the problem. It is an artifact of starting from the wrong assumption.

10.2 Why Filesystem-First Works

The filesystem is the oldest, most tested, most universally available abstraction in computing. Every operating system provides it. Every programming language can interact with it. Every tool — from `cat` to `rsync` to `grep` — operates on it. No SDK is required.

By making the filesystem the sole source of truth for agent state, we eliminate entire categories of problems:

No connection state. There are no sessions to manage, no connections to keep alive, no heartbeats to maintain. An agent that crashes and restarts finds all its messages in its inbox directory, exactly where the daemon left them.

No database. Messages are text files. The directory listing is the index. Sorting by filename gives chronological order. Searching by content is `grep`. Backup is `cp -r`. Migration is `mv`.

No deployment. To add an agent to the system, create a directory and a configuration file. To remove an agent, delete the directory. To move an agent to a different machine, copy the directory.

Universal tooling. System administrators can monitor agent communication with `tail -f inbox/`. Developers can debug message delivery with `ls -la`. No specialized client is needed.

10.3 Why Transport Independence Matters

Networks fail. Protocols change. APIs are deprecated. Cloud services are discontinued. But files persist.

An agent that communicates via the AgentTalk relay today can switch to email tomorrow by changing one line in its configuration. Transport independence decouples the protocol's longevity from any single transport's lifespan. SMTP has been operational since 1982 [Postel, 1982]. FTP's origins date to 1971 [Bhushan, 1971], with the current standard (RFC 959) published in 1985 [Postel and Reynolds, 1985]. HTTP since 1991 [Fielding et al., 1999]. A protocol built on all three inherits the survivability of all three.

10.4 Why Identity Must Live in the Message

Most security architectures place identity at the transport layer: TLS certificates, OAuth tokens, API keys. This works when all communication traverses a single transport. It fails the moment a message crosses a transport boundary.

Ed25519 inline signatures solve this by attaching identity to the message itself. Our experiment validated this design: 1,744 signed messages traversed HTTPS, SMTP, and FTP without any signature being invalidated.

10.5 The UNIX Philosophy Applied to AI

The design principles of this system — small tools, text streams, composability — are the UNIX philosophy [McIlroy et al., 1978, Kernighan and Pike, 1984]:

- Write programs that do one thing well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

The `agentazall` CLI follows this philosophy precisely. `send` sends a message. `inbox` lists messages. `read` reads a message. Each command does one thing. They compose via the filesystem.

The UNIX philosophy scales. The evidence is the internet itself — built on small, composable protocols (TCP, DNS, SMTP, HTTP) rather than monolithic architectures. We argue that agent communication should follow the same design trajectory.

10.6 Scalability Without Connection State

Traditional client-server protocols scale poorly because each client requires server-side state. The filesystem-first approach eliminates connection state entirely. The relay server maintains no sessions, no connection pools, no per-agent state beyond the bearer token hash. Adding an agent means creating a subdirectory. The relay itself is stateless by design: it holds messages transiently in RAM and requires no database, no connection tracking, and no per-user session management. Scaling the relay is a matter of adding capacity, not managing complexity.

This does not mean the system handles all scaling challenges. Message delivery latency increases with agent count because the daemon polls sequentially. Real-time streaming is not supported. But for asynchronous message exchange with response times measured in seconds, the filesystem model provides sufficient throughput with minimal infrastructure.

10.7 Beyond Language Models — A Universal Service Layer

The protocol's properties — endpoint-agnostic addressing, transport independence, whitelist-based access control — make it applicable to a broader class of AI services. Every model, tool, or service on a local network becomes addressable through the same mechanism. Address filtering provides access control without infrastructure.

To validate this claim, we built three non-LLM utility agents on the same protocol: a translation service (NLLB-200), a speech-to-text service (Whisper), and a text-to-speech service (Kokoro TTS). Binary attachments were validated to survive the AgentTalk relay transport byte-for-byte — a 32 KB WAV file and a 69-byte PNG both arrived with identical SHA-256 checksums. No protocol modifications were required.

10.8 Limitations

This protocol does not attempt to solve every problem in agent communication:

No real-time streaming. The daemon polls at configurable intervals (default: 3 seconds). For applications requiring sub-second communication, this protocol is inappropriate.

No structured tool calling. Tool invocation must be expressed in natural language within the message body.

No message ordering guarantees. Messages are ordered by filesystem timestamp. The protocol does not provide sequence numbers or vector clocks.

No group semantics. The protocol supports multi-recipient messages (via CC headers) but does not define group membership or permissions.

Filesystem as abstraction, not requirement. The reference implementation uses a POSIX filesystem, but the protocol depends on the *semantics* of a filesystem — named entries, hierarchical directories, read, write, list — not on any particular storage substrate. Any backend providing these operations qualifies: an in-memory key-value store, a SQLite database, a cloud object store, or a browser's IndexedDB.

Security is deployment-dependent. The protocol provides message-level authentication (Ed25519) and address-based filtering, but deployment-level choices determine whether these are enforced. The protocol provides the tools; the deployment provides the policy.

These limitations are deliberate. Each represents a design decision to keep the protocol simple rather than comprehensive.

11 Conclusion

11.1 Summary

We presented a filesystem-first communication protocol for autonomous AI agents that achieves transport independence, model independence, and cryptographic identity through a design of deliberate simplicity. A message is a text file. A mailbox is a directory. Identity is an Ed25519 keypair. The transport is a pluggable adapter that the agent never sees.

We validated this design empirically. Four autonomous LLM instances — spanning three distinct model architectures (Qwen3-Coder-Next 81B, Hermes-4-70B, Devstral-Small 24B) — exchanged 1,744 cryptographically signed messages across three transport protocols (HTTPS relay, SMTP/IMAP email, FTP) in 30 minutes of unattended operation. All transports delivered reliably. All signatures survived transport transitions. All models comprehended and responded to each other's messages without any form of capability negotiation, schema alignment, or protocol adaptation.

The protocol's overhead was measured at under 5 ms per message for local transports and under 200 ms for relay transport — less than 10% of the total cycle time in all configurations. Inference latency, not communication latency, was the dominant factor in every round.

Extended real-world usage over multiple weeks — involving Claude Opus 4, Qwen3.5-9B, and Devstral-24B instances communicating across the public internet — provided observational evidence that the protocol sustains coherent multi-turn collaboration in uncontrolled environments.

11.2 The Thesis Restated

The question that motivated this work was deceptively simple: what is the minimum viable communication protocol for autonomous AI agents?

The answer turned out to be: less than anyone expected.

No session management. No capability negotiation. No structured schemas. No service discovery. No connection pools. No task lifecycle. No webhook infrastructure. No cloud dependency. Just files, directories, and a daemon that moves them.

This is not a claim that existing protocols are wrong. MCP, A2A, and ACP solve real problems for their target deployments. But they solve those problems by adding machinery — and that machinery has costs: complexity, fragility, coupling, and the assumption of perpetual connectivity. Our contribution is the demonstration that for the specific case of asynchronous, loosely coupled agent messaging, much of that machinery can be avoided.

The protocol's design can be summarized in three sentences: Messages are text files with inline signatures. Transports are interchangeable adapters. The filesystem is the only required infrastructure.

11.3 Contributions

This work makes the following contributions:

- **A working, open-source protocol** for filesystem-first agent communication, implemented in Python with zero mandatory dependencies, supporting three transport backends and Ed25519 cryptographic identity.
- **Empirical validation** of cross-model, cross-transport agent communication at scale, demonstrating that architecturally distinct LLMs can sustain coherent multi-party conversation through a plain-text protocol.
- **A minimal MCP integration pattern** (the "doorbell pattern") that provides push notification for MCP-aware clients without coupling the messaging layer to the MCP session lifecycle.
- **Quantitative analysis** of protocol overhead versus inference cost, establishing that communication protocol overhead is negligible ($< 10\%$) compared to LLM inference time for short-form agent messages.
- **Evidence that transport independence is achievable** through message-level identity, demonstrating that Ed25519 inline signatures survive serialization across HTTPS, SMTP, and FTP without modification.

11.4 Future Work

Structured attachments. A lightweight attachment type system — without imposing structure on the message body — would extend the protocol's utility for tool-mediated workflows.

End-to-end encryption. Adding X25519 key exchange for per-message encryption would enable private communication across untrusted transports.

Relay federation. A federation protocol where relays discover each other and forward messages for non-local recipients would provide email-like resilience.

Formal verification. A formal specification suitable for automated verification of message delivery guarantees and deduplication correctness.

Large-scale agent populations. Experiments with tens to hundreds of agents across multiple relay servers would establish the practical scaling boundaries.

Heterogeneous endpoint validation. A larger-scale follow-up experiment deploying a mixed network of language models, diffusion pipelines, and utility services under sustained load.

Collaborative research methodology. We intend the next iteration of this research to be conducted using the protocol itself. Human researchers and AI agents, communicating

through the filesystem-first message format, will collaboratively design, execute, and write up experiments on the protocol's evolution.

11.5 Artifact Availability

All artifacts described in this paper are publicly available [Koch, 2025]:

Artifact	Location
Source code	https://github.com/cronos3k/AgentAZAll
Python package	https://pypi.org/project/agentazall/ — `pip install agentazall`
Live demo	https://huggingface.co/spaces/cronos3k/AgentAZAll
Public relay	`relay.agentazall.ai` — open for testing
Project website	https://agentazall.ai

The integration test can be reproduced using `run_integration_test.py` included in the repository.

11.6 Closing Remark

The history of computing is a history of rediscovering simplicity. The internet succeeded not because it was the most sophisticated network architecture, but because it was the simplest one that worked. Email succeeded not because it was the best messaging system, but because it was the most universal one. UNIX succeeded not because it was the most powerful operating system, but because it was the most composable one.

We believe agent communication is at a similar inflection point. The current generation of protocols is sophisticated, capable, and complex. But the agents themselves — language models with the ability to read, reason, and write — do not need sophisticated protocols. They need text, a place to put it, and a way to find it.

Four agents. Three architectures. Three transports. 1,744 messages. Every message signed. Every signature verified. Every transport interchangeable. Zero downtime. Zero schema negotiation. Zero cloud dependency. A filesystem, a daemon, and plain text.

That is what we built. And it works.

References

A2A Project, Linux Foundation. Agent2agent protocol (A2A) specification. https://a2a-protocol.org/latest/, 2025. Originally released by Google, April 2025; donated to the Linux Foundation, June 2025.

Anthropic. Model context protocol specification (2025-11-25). https://modelcontextprotocol.io/specification/2025-11-25, 2024. Donated to the Agentic AI Foundation under the Linux Foundation, December 2025.

Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

Abhay Bhushan. File transfer protocol. RFC 114, Network Working Group, April 1971.

Mark Crispin. Interactive mail access protocol — version 2. RFC 1064, Internet Engineering Task Force, July 1988.

Mark Crispin. Internet message access protocol — version 4rev1. RFC 3501, Internet Engineering Task Force, March 2003.

Roy Fielding, Jim Gettys, Jeffrey Mogul, et al. Hypertext transfer protocol — HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM '94)*, pages 456–463. ACM, 1994.

Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Technical Report SC00061G, 2002.

Gerganov Georgi et al. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023.

IBM Research. Agent communication protocol (ACP). https://agentcommunicationprotocol.dev/, 2025. Merged into A2A under the Linux Foundation, August 2025.

Brian Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984.

Gregor Koch. AgentAZAll: Filesystem-first agent communication. https://pypi.org/project/agentazall/, 2025.

Yingxuan Li, Jie Song, Jiaxing Hit, et al. A survey of agent interoperability protocols. arXiv:2505.02279, 2025.

Linux Foundation. Linux foundation announces the formation of the agentic AI foundation. https://www.linuxfoundation.org/press/linux-foundation-announces-the-formation-of-the-agentic-ai-foundation, December 2025.

M. Douglas McIlroy, E. N. Pinson, and B. A. Tague. UNIX time-sharing system: Foreword. *The Bell System Technical Journal*, 57(6):1899–1904, 1978.

Mistral AI. Devstral: Development-focused language models. https://huggingface.co/mistralai/Devstral-Small-2505, 2025.

NousResearch. Hermes 4: Conversational language models. https://huggingface.co/NousResearch/Hermes-4-70B, 2025.

OpenAI. Agents API and SDK documentation. https://developers.openai.com/api/docs/guides/agents/, 2025.

Jonathan Postel. Simple mail transfer protocol. RFC 821, Internet Engineering Task Force, August 1982.

Jonathan Postel and Joyce Reynolds. File transfer protocol (FTP). RFC 959, Internet Engineering Task Force, October 1985.

Qwen Team. Qwen3-coder-next: Hybrid architecture language model for code. https://huggingface.co/Qwen/Qwen3-Coder-Next, 2025.