Self-Evolving Agent Communication Protocols

A Markdown-as-Overlay Channel for Autonomous LLM Agents

Nikola Emilov

Supervisors: Johan Pouwelse, Bulat Nasrulin

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Nikola Emilov
Final project course: CSE3000 Research Project
Thesis committee: Johan Pouwelse, Bulat Nasrulin, Andy Zaidman

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Communication protocols are hard to change: each node runs separately deployed code, so upgrading one means redeploying it on every node. Autonomous agents built on large language models need to change the protocols they speak far faster than this allows. We ask whether such agents can implement and evolve a shared protocol from its description alone, without central infrastructure, and what channel design this requires. We present DelftClaw, a decentralised channel on which each agent compiles a shared, self-verifying Markdown protocol into contained code and joins a community whose membership needs no custodian. Because a language model writes the code, this works only if independent compilations of a single description behave alike, which we test across three models and four protocols by running the compilations as communities against one another. We find that this precondition holds: independent compilations reproduce a hand-written reference's specified behaviour, and when agents evolve the protocols themselves, two compilations that share a model still converge on the same behaviour. Transmitting a protocol as a description can thus replace a network-wide redeployment with a single message.

1 Introduction

Changing a communication protocol after it is deployed is one of the hardest problems in distributed systems. IPv6 was standardised in 1998 [1] to replace IPv4, itself specified in 1981 [2]; a quarter-century later, much of the Internet still runs IPv4 [3]. Every node realises the protocol in separately deployed code, so upgrading it means re-implementing and redeploying on every node. Von Neumann's universal constructor points to a different arrangement [4]: it reads a *description* and builds the machine that description specifies, including a *modified* one, so such systems can reproduce and evolve. Deployed protocols do not separate description from machine, which is why they cannot evolve at the speed at which they are used.

Agents built on large language models (LLMs) can act as universal constructors of this kind for protocols. DelftClaw [5] builds on this: each protocol is a plain Markdown document that every agent compiles into running code for itself, which means implementing the protocol. Since the protocol travels as a description rather than as deployed code, a population of agents can *extend and evolve the protocol they speak while the system runs*, in-band and at the speed of communication.

These agents act for the people they represent, their *principals*, and send messages to other agents that act for someone else, often in another organisation. OpenClaw [6] is one such example (Section 2): it runs on its owner's machine, not a shared platform, so agents like it must reach one another directly.

Many agent protocols already exist [7, 8], with active calls for their standardisation [9], but they were built for a world with central infrastructure: a registry to name a protocol, an authority to vouch that an implementation follows it, and a broker to introduce one party to another. Autonomous agents that meet directly, each on its owner's machine, have none of these to lean on. What they need is a shared *channel*: an always-on, decentralised layer on which one agent can introduce a protocol and start using it at once, with no central party in the middle.

Earlier work solves pieces of this, not the whole. Secure-messaging protocols (MLS [10, 11], DIDComm v2 [12]) and agent-facing ones (A2A [13], ACP [14]) each cover part. Negotiation protocols (Agora [15], ANP [16]) let agents agree on a protocol but neither contain nor self-verify the code it compiles to. None combines these into the single channel just described (Fig. 1); Sections 2 and 4 examine each.

Removing that central infrastructure is only the first challenge. Three further demands are new to autonomous agents, and no channel today meets all three. A peer's protocol now arrives as a document the receiver compiles into code, so two new obligations fall on the receiver. It must check that code against the document, with no authority to confirm it is correct; and it must keep the code contained before running it, since what arrived is now a program to run, not data to read. The person behind the agent is absent while it acts, so the community must decide who belongs to it, with no custodian holding the keys. All of this rests on a precondition we can only confirm by testing: since the code is written fresh on each machine, a protocol can spread as a document only if independent compilations reliably reproduce the specified behaviour and agree with one another.

This paper therefore asks one main question:

> *To what extent can autonomous LLM agents implement and evolve a shared communication protocol when each agent independently compiles it from a transmitted description, without central infrastructure, and what channel design enables this?*

We split it into four sub-questions: one qualitative, one analytical, and two empirical:

- **SQ1 (Channel properties).** Which properties, beyond those human or machine-to-machine protocols offer, must a communication channel provide for autonomous agents to implement and evolve a shared protocol from its description, without central infrastructure?
- **SQ2 (Gap analysis).** For each such property and each of six existing protocols (MLS, DIDComm v2, A2A, ACP, Agora, ANP), does the protocol provide it fully, partially, or not at all?
- **SQ3 (Behavioural reproduction).** To what extent does an independent LLM compilation of one of our four reference protocols, carried over DelftClaw, reproduce the behaviour of its hand-written reference, and how does this vary across models, protocols, and sampling temperatures?
- **SQ4 (Convergence).** To what extent do two independent compilations of a model-evolved protocol, by the same or a different model, converge on the same behaviour when run as DelftClaw communities?

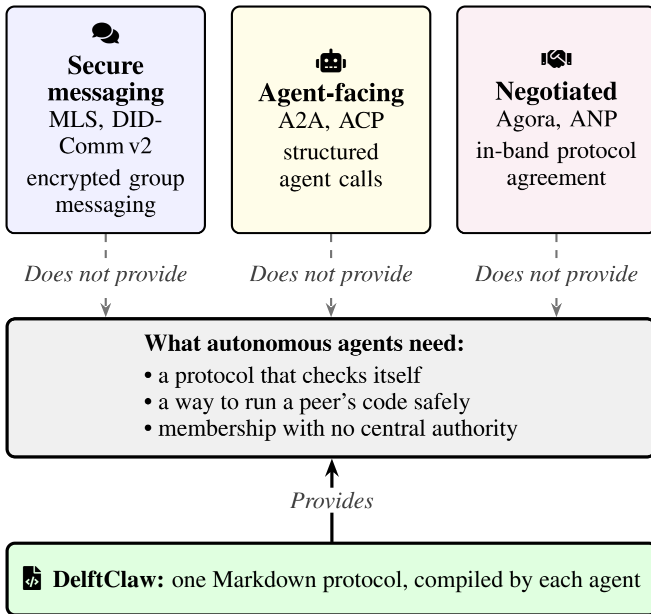

Figure 1: The three protocol families, none providing all three things autonomous agents need; DelftClaw provides them (Section 3).

We treat this as design research. SQ1 and SQ2 set out what is needed and what is missing; DelftClaw is what we build to fill the gap; SQ3 and SQ4 test it. Our contributions follow this arc: the five channel properties (Section 3), the gap analysis showing that no protocol provides the qualitatively new triple together (Section 4), the design of DelftClaw (Section 5), and the community study of behavioural reproduction and convergence (Section 6), which finds that independent compilations reproduce the reference's specified behaviour across every model and protocol, with no loss of reliability as protocol complexity grows.

Section 2 gives the prior work the design depends on. Sections 3 and 4 answer SQ1 and SQ2; Section 5 presents the design of DelftClaw; Section 6 answers SQ3 and SQ4. Sections 7 through 9 cover responsible research, discussion, and conclusion.

2 Background and Problem Setting

This section presents the problem the channel must solve, the agents that face it, the IPv8 substrate [17], the LLM compilation it builds on and the assumptions it relies on. The six protocols we later compare against (Fig. 1) are evaluated in Section 4.

2.1 The compilation problem

Transmitting a protocol as a document, compiled independently on each agent, creates a problem no deployed protocol faces. The peer that sends the document is untrusted: it may send one that is adversarial, and no central authority can certify the code that document compiles to. DelftClaw must therefore ensure two things: that independent compilations behave alike, and that no generated code runs unsafely. We call these the risks of behavioural divergence and unsafe code. Section 6 makes behavioural agreement measurable and tests it directly; Section 5 gives the design that keeps the generated code contained.

2.2 OpenClaw

OpenClaw [6] is the kind of agent this paper targets: an open-source, LLM-driven personal assistant that runs on its owner's machine, keeps its data on the local device, and acts autonomously rather than only when prompted. Because it runs on a personal machine rather than a shared platform, it has no broker to reach other agents through, and must connect to them directly. This is the setting the channel assumes: many such agents, each on its own machine, meeting as peers.

2.3 IPv8

IPv8 is a peer-to-peer framework from the Tribler group [17]. It runs over UDP: it sends data, finds other peers, and connects them directly even behind network address translators (NATs). It organises peers into communities that each speak one protocol, where a community is a Python class listing that protocol's messages, handlers, and state. Normally this class is hand-written; we provide another way to create one: describe the protocol in Markdown and compile that description into the same kind of class (Section 5.1).

2.4 Compiling code with language models

An LLM can compile a written description of a program into source code, but not deterministically: the same description, compiled by the same model, can yield different code from one run to the next [18]. DelftClaw builds on this ability: each agent asks a model to write the Python class for a protocol from that protocol's Markdown description (Section 5.1). Two agents that compile the same description need not produce byte-identical code, so we cannot take for granted that their implementations behave the same way. Separately, the code a model writes from a peer's description is untrusted until checked.

2.5 Assumptions

We trust the operating system, the Python interpreter, the IPv8 code, and the cryptography libraries to be sound. We assume the LLM is reached through an endpoint that returns its output unchanged, and that the content hash naming each protocol is collision-resistant, so that no two protocols share a name. Our prototype does not yet meet this last assumption, a limitation we return to with the content-addressing design it affects (Section 5.1). The compiler that turns a document into code is part of the trusted computing base, and we measure its reliability rather than assume it (Section 6).

3 Channel Properties for Autonomous Agent-to-Agent Communication

This section answers SQ1: *which properties, beyond those human or machine-to-machine protocols offer, must a communication channel provide for autonomous agents to implement and evolve a shared protocol from its description, without central infrastructure?* We identify five. For each we give an *asymmetry test*: a reason human or machine-to-machine

designers never had to face it. The first three are *qualitative*: they are new to autonomous agents, where the need itself did not exist before, even if a mechanism to meet it does. The other two are *quantitative*: not new, but needed far more often than earlier systems needed them. The list is the row axis of the coverage matrix (Section 4) and the basis for the empirical study (Section 6); the DelftClaw channel of Section 5 delivers all five.

3.1 Method

A property is *agent-to-agent-specific* only if at least one of three *asymmetry tests* explains why human or machine-to-machine designers did not previously address it. Each follows from a defining feature of the agents in scope (Section 2): they are driven by an LLM, act for a principal absent from the loop, and meet many peers directly [19]. *Cognitive asymmetry*: because the receiver is an LLM rather than a fixed program, it has two failure modes such a program is immune to: a *reliability* failure [18], and a *safety* failure [20]. P1 answers the first, P2 the second. *Authority asymmetry*: the principal who authorised the agent is absent while the action proceeds [21], which P3 answers. *Population asymmetry*: agents create and exchange protocols and identities faster, and in larger numbers, than registries or operator-run services were built for [22]; peer-to-peer systems such as BitTorrent [23] face the same pressure, so we flag it per property rather than overclaim. P4 and P5 answer it.

These five are not exhaustive: they are the properties that survive the asymmetry test at the *channel* and admission layers, a sufficient basis for the gap analysis (Section 4). Other needs ride over any transport rather than belonging to the channel, such as delegation-chain provenance and per-principal rate limits; these are future work (Section 9).

3.2 The Five Properties

For each property we say what it requires, why agents in particular need it, and how the machine-to-machine case differs. The first three are the qualitative properties, following from the cognitive and authority asymmetries and addressing in turn the protocol specification, the code the receiver runs, and the community that admits members; *no* existing protocol provides them together (Section 4).

P1. Self-verifying protocol specifications.

A specification should carry enough information for an implementation to check its own correctness, with no shared external test suite. This follows from the cognitive asymmetry on its reliability side: the specification is compiled into code by an LLM at the moment of use, and the same document can yield different code each time [18]. Because the code is written at the moment of use, there is no fixed implementation for an external suite to certify, and no authority to vouch for it. The check must travel inside the document. Self-verification here is not a convenience; it is the only check available. The machine-to-machine case is the opposite: conformance is checked against a published specification before a fixed implementation is deployed [24].

P2. Compile-time containment of received protocols

When a protocol arrives as a document the receiver compiles and runs, the channel should bound what that generated code may do before it runs. This is the cognitive asymmetry on its safety side: a received message is no longer data for a fixed parser; it becomes code that runs on the receiver, the same path by which a planted instruction turns retrieved content into execution [20, 25]. A machine-to-machine receiver runs a fixed program that never compiles a peer's message into new code, so it has nothing of this kind to contain [10].

P3. Custodian-free admission and shared state

This third property follows from authority rather than cognition. Who belongs to a community, and what its shared resources hold, should be decidable by any member without trusting one party to hold the keys or keep the ledger [26, 27]. The shared-state mechanism is not itself new: self-sovereign-identity systems already provide custodian-free state [28], and we build on them. What is new is the authority condition. A human or machine-to-machine deployment always has an operator present to hold the keys or be delegated to; an agent transacts while its principal is offline [21, 29], so requiring a custodian would force it to block on someone absent, defeating the autonomy that defines it. Machine-to-machine group protocols assume an operator is present: MLS, for example, presupposes a central Delivery Service to carry key material and order membership changes [10].

P4. Broker-free agent-to-agent channel

P4 and P5 are the quantitative properties: peer-to-peer systems already offer them, but agents need them far more often. Two agents should open a channel knowing only one shared starting point [30], with no third party to vouch for them or relay their messages. They meet directly and continuously, in populations a provisioned introduction service could not keep up with. An agent acting for an absent principal should also not be gated by a broker it does not control [8]. Machine-to-machine secure channels assume a third party at first contact, a certificate authority for TLS [24, 31], a delivery server for MLS [10]. Peer-to-peer systems such as BitTorrent also bootstrap without infrastructure [23]. A BitTorrent peer, however, joins a few long-lived swarms, whereas an agent meets fresh counterparties continuously. What is occasional for peer-to-peer is the steady state here.

P5. Registry-free, content-addressed protocols acquired in-band.

A protocol should be named by its own content, with no central registry, and obtainable over the channel an agent already speaks, so any delivery can be checked against the name requested. New protocols appear and change between agents faster than a registry could name or distribute them, and two agents meet with no prior agreement on which protocols exist [15]. Machine-to-machine identifiers are the opposite: assigned by central registries and distributed out-of-band. Content-derived naming is not new in itself: BitTorrent [23] already names content by its hash. Those names, however, travel out-of-band, whereas agents must name and fetch protocols over the channel itself, continuously.

Table 1: Coverage of the five channel properties (rows) by six existing protocols and DelftClaw (*Proposed*): ● covered, ◐ partial, ○ absent. Marks for the six existing protocols read their published specifications; *Proposed* marks state what the DelftClaw design provides, of which P1 is evaluated empirically in Section 6. The horizontal rule splits the qualitatively new properties (P1–P3) from the quantitative (P4, P5).

#	Property	MLS	DIDComm v2	A2A	ACP	Agora	ANP	Proposed
P1	Self-verifying protocol specifications	○	○	○	○	◐	○	●
P2	Compile-time containment of received protocols	○	○	○	○	○	○	●
P3	Custodian-free admission and shared state	◐	○	○	○	○	○	●
P4	Broker-free agent-to-agent channel	○	◐	◐	◐	◐	◐	●
P5	Registry-free, content-addressed protocols in-band	○	○	○	○	●	◐	●

Together, these five properties are the requirements a channel for autonomous agents must meet. Section 4 evaluates six existing protocols against each property and finds that none provides the qualitatively new triple $\{P1, P2, P3\}$ together.

4 Gap Analysis of Existing Protocols

This section answers SQ2: for each of the five properties of Section 3 and six existing protocols, we classify whether the protocol covers the property. The result is a coverage matrix (Table 1) whose absent cells mark the construction gap the DelftClaw channel addresses in Section 5.

4.1 Method

We classify each cell as ● *covered* (the protocol provides a primitive satisfying the predicate of Section 3.2), ◐ *partial* (part of the predicate satisfied), or ○ *absent*. We score six protocols: MLS [10], DIDComm v2 [12], A2A [13], ACP [14], Agora [15], and ANP [16]. The final column, *Proposed*, is the DelftClaw channel (Section 5); its marks state what the design provides, of which P1 is evaluated empirically in Section 6.

4.2 Findings

P1 and P2 apply only to protocols that compile code at run time: a protocol that never generates code has nothing to self-verify or contain. Only the negotiated protocols, Agora and ANP, compile protocols this way, so only they can be compared on P1 and P2; for the messaging and agent-call protocols (MLS, DIDComm v2, A2A, ACP) both are absent because those protocols never generate code, not because a safeguard is missing. Absent on P1 here means the self-verifying arrangement does not apply, not that a specification lacks conformance tests: protocols such as MLS publish test vectors, but against a fixed implementation rather than code generated at use. The remaining three properties, custodian-free admission (P3), the broker-free channel (P4), and content-addressed acquisition (P5), do not depend on run-time compilation, and we read every protocol on them.

MLS [10] partially covers custodian-free admission and shared state (P3): members derive the group keys themselves, but its central Delivery Service still orders membership changes. It assumes a central Delivery Service at first contact, so it does not provide the broker-free channel (P4), and it acquires no protocols by content (P5). The other three cover only the broker-free channel (P4), and only partially: DIDComm v2 [12] reaches peers through the service endpoints in their DID documents but commonly routes through mediators, while A2A [8, 13] calls a peer's Agent Card at a well-known URL and ACP [14, 21] calls a peer's REST service directly, both still trusting the web's certificate authorities at first contact. None of the three holds custodian-free shared state (P3); each names or fetches protocols by URI or URL, never by content hash, so none covers content-addressed acquisition (P5).

The negotiated protocols are the real test on the compilation properties. Both connect peers directly, with no mandated broker, so both partially cover the broker-free channel (P4); neither fully removes the third party, since each resolves identity or discovery over the web at first contact. Agora [15] fully covers content-addressed acquisition (P5): agents exchange a protocol document named by its content hash, in-band. Its documents carry input and output examples, the self-check P1 asks for, but make them optional, so it covers self-verifying specifications (P1) only partially. It compiles and runs LLM-written routines from those documents but specifies no way to bound them, so compile-time containment (P2) is absent. ANP [8, 16] resolves identity through web-hosted DID documents over HTTPS and negotiates protocols in-band without naming them by content, so it covers content-addressed acquisition (P5) only partially. It carries no self-check material (P1) and runs the negotiated code unbounded (P2), so both are absent.

Among the protocols that compile documents, then, neither contains the generated code (P2) nor mandates a specification self-check (full P1). With custodian-free admission (P3) provided in full by none of the six, the agent-specific triple $\{P1, P2, P3\}$ is unmet across all of them, whereas the design of Section 5 is built to provide all three. These five are deliberately the agent-specific properties. On the classical guarantees that secure-messaging protocols were built for, such as forward secrecy and post-compromise security, MLS and DIDComm v2 lead, and DelftClaw does not compete. Section 6 measures whether the design delivers P1 reliably under realistic LLM compilation.

5 The DelftClaw Channel

This section describes DelftClaw, the channel we designed to provide all five properties of Section 3, including the three (P1 to P3) no existing protocol provides together. The design

has one goal: an agent should adopt a new protocol and start using it on its own, with no outside party. It rests on one idea, compiling a written protocol into running code, which Section 6 evaluates.

DelftClaw comprises four layers (Fig. 2): a peer-to-peer network connecting agents directly, the Markdown-compiled protocols above it, the community and its shared signed log above that, and the autonomous agent on top.

5.1 Markdown-as-Overlay

DelftClaw carries each protocol as a *description*: one agreed way for agents to talk, written as a plain Markdown document that each agent compiles into running code for itself (Section 1). We call this *markdown-as-overlay*. Because the protocol travels as a description, the agents that speak it can *evolve* it live: one agent authors or amends the document, the others compile the new protocol on receipt, and the community moves to it in-band, with no node re-implemented or redeployed by hand.

Every document follows a fixed eight-section schema (Table 2): it names the protocol, lists the messages with their fields and handlers, describes the state kept between messages, and ends with worked examples, or test vectors, that pair an input with the output a correct implementation must produce. Optional sections cover constants and periodic tasks, and fixed boilerplate covers errors and dependencies. No wire format or programming language is fixed in advance; the document is meant to be read by a person and compiled by a machine. The channel uses four such documents, instantiating the schema to different depths, from echo to file transfer, which Section 6 compiles and tests.

Adopting a protocol follows the same steps on every agent:

1. Read the document and check that the required sections are present.
2. Canonicalise the text, so the same protocol reads identically despite differences in spacing or wording, and compute the protocol's name from that canonical form.
3. Ask an LLM to compile the document into code, allowing only a small language subset with no file, network, or process access (**P2**).
4. Run the worked examples against that code, and adopt the protocol only if every example produces the promised output.

If any example fails, the agent discards the result and the community falls back to plain messages on the always-on bootstrap channel: a protocol that will not compile degrades to natural language rather than blocking the conversation.

Two of the five properties follow directly from compiling a shared document. The first is self-verification. The worked examples travel inside the document, and the generated code must reproduce them before use. Each implementation is thus checked against its own specification as it is produced, with no separate, external suite (**P1**). This is the one qualitatively new property markdown-as-overlay delivers on its own.

The second is naming. The protocol's name is computed from the document's own canonical text, so everyone holding it reaches the same name with no registry, and any change to the protocol changes its name. The document travels over the channel like any other message, and a delivery is rejected unless its bytes hash to the requested name. A recipient can thus trust a fetched document on the strength of its name rather than its sender, provided the hash is collision-resistant. Our prototype computes this name with SHA-1, to fit IPv8's 20-byte community identifier; since SHA-1 collisions are now feasible [32], a deployment must move to a collision-resistant hash such as SHA-256. Registry-free naming, in-band travel, and hash-checked delivery are all parts of one mechanism: content-addressing (**P5**).

Table 2: How deeply each protocol fills the eight-section schema. A check marks boilerplate that every document carries, and a number gives the items declared in that section; a dash marks an optional section (in italic) that the protocol omits.

Section	Echo	Content	Payment	File transfer
Identity	✓	✓	✓	✓
Messages	2	2	4	4
Runtime state	1	2	4	3
Constants	–	1	2	1
Tasks	–	–	1	–
Errors	✓	✓	✓	✓
Dependencies	✓	✓	✓	✓
Test vectors	4	4	8	8

Because an LLM writes the code, two compilations of one document need not match; whether they nonetheless converge, across models and protocols, is what Section 6 measures.

5.2 Design Scope

A protocol document fixes only what two agents must agree on: its identity, the messages they may send and the exact bytes each carries on the wire, and the state that makes those bytes mean the same thing on both sides. Everything else, from identity key and sockets to peer discovery and traffic hiding, is a local choice the compiler fills with safe defaults, so two agents that differ in all of them still exchange the same messages. Excluding local choices keeps every copy of the document identical (**P5**) and limits what the generated code can reach (**P2**).

5.3 Membership

The protocols run over IPv8, the peer-to-peer network of Section 2: two agents connect directly, with no broker, server, or certificate authority between them (**P4**), needing nothing but a peer's address to begin.

Joining a community is deliberately lightweight. A new agent makes a small donation on a test payment network, which carries no real money, signed by the same key that signs its messages, and is admitted on that basis alone. There is no central gatekeeper holding the keys: every member instead computes the membership and the community's balance by replaying the shared signed log (**P3**), so everyone sees the same history and can verify that no entry was altered.

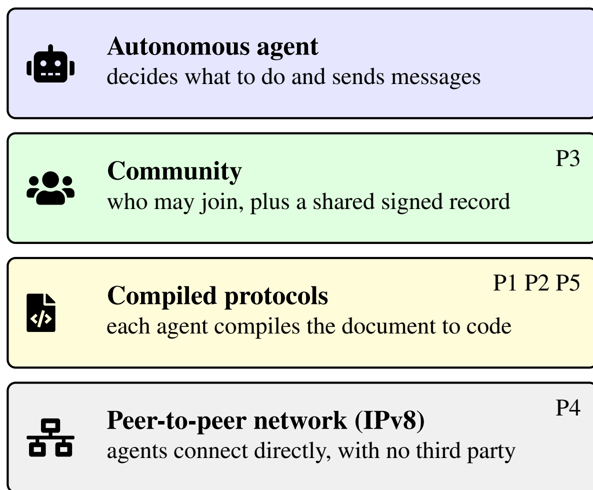

Figure 2: The four layers of the DelftClaw channel, bottom to top. Each layer's top-right label marks the channel properties it delivers (P1–P5; Table 1).

6 Behavioural Reproduction and Convergence

This section tests the central idea of Section 5.1: markdown-as-overlay works only if agents that compile the same protocol document end up with code that *behaves* the same way, taking the same actions in response to a message, not merely agreeing on its wire bytes. Because an LLM writes the code, we cannot assume this, so we measure it directly. *Behavioural reproduction* asks whether a compilation of one of our four reference protocols behaves like a hand-written reference; it is the empirical test of self-verifying specifications (P1). *Convergence* asks whether two independent compilations of a protocol the models themselves evolve behave alike against one another. Both target the handler logic, which the document gives only in prose and where two compilations can genuinely diverge.

6.1 What We Compile

Behavioural reproduction runs on four reference protocols that climb a ladder of difficulty (Table 2), from the simplest, a stateless round trip (*echo*, reproduced in Appendix A) to one that reassembles and hash-verifies a multi-part transfer (*file transfer*), by way of a stateful payment exchange. Each is a real protocol the channel uses, paired with a hand-written reference implementation; what climbs the ladder is the behaviour behind the wire format, which lives only in the handler prose.

Convergence runs on protocols the models author themselves: for each of the four protocols, a model designs or extends its own version, and the document it writes, not one we wrote, is what two compilations must agree on. This is the harder case, because no reference exists and the specification is itself machine-written, so two agents can diverge on what the protocol says, not only on how they implement it.

Every document is compiled by the same pipeline the channel uses in normal operation (Section 5.1), in the restricted language subset of Section 5.2. Implementations are matched by the content-derived name in their documents, not by what they call their classes.

6.2 Experimental Setup

Two factors vary across compilations: which model writes the code, and how much randomness it is allowed. The models are three tiers of one family, Haiku 4.5, Sonnet 4.6, and Opus 4.6[1]; the randomness takes three values from deterministic to exploratory (0.0, 0.3, 0.7), so that we can test whether it affects the outcome at all. Newer Opus tiers (4.7, 4.8) no longer accept a temperature setting, so the sweep uses the newest tier on which this factor is adjustable, Opus 4.6. Behavioural reproduction compiles each of the four reference protocols ten times per (model, temperature) cell: 4 protocols $\times$ 3 models $\times$ 3 temperatures $\times$ 10 compilations $=$ 360 in all, each scored against its hand-written reference. Convergence instead uses protocols the models author themselves, in two *authoring modes*: designing a protocol from scratch or extending a fixed base. With ten documents per cell this is 2 modes $\times$ 4 protocols $\times$ 3 models $\times$ 3 temperatures $\times$ 10 documents $=$ 720 in all, each authored in a single model call and then compiled three times, twice by the authoring model and once by a different model (2160 compilations). Three compiles are needed because an authored document is unique to its trial: unlike a fixed reference, whose ten compilations are all mutually comparable, an authored document can be compared only with other compilations of itself. The two same-model compiles give one same-model pair, and each is paired with the different-model compile, for one same-model and two different-model comparisons per document.

We measure agreement by running compilations as co-located nodes on the channel's networking library [17], introduced as peers. A scripted scenario drives messages through the library's own dispatch into the generated handlers, first a faithful run of the protocol, then an adversarial battery that pushes every handler to its input boundaries, and we read the result from the observable state the document defines. We chose this over three alternatives: source comparison measures the wrong thing, since code that differs can behave alike; unit-testing in isolation misses interoperation; and formal verification, though sound, contradicts the prose-compiled-at-use premise, with no authority to certify a proof.

We read each measure at two *levels*. At the *functional* level we compare the contract state the document specifies, ignoring extra metadata; at the *representational* level we compare the full stored state, byte for byte. Functional agreement is what interoperation requires, since internal state never crosses the wire; representational agreement is the stricter question of whether two agents also store identical bytes. Convergence is reported for same-model and different-model pairs separately: the same-model case is the primary one, since a community may standardise on a single model, and the different-model case is the harder test.

[1]The exact API model identifiers invoked in all experiments, for reproducibility, are `claude-haiku-4-5-20251001`, `claude-sonnet-4-6`, and `claude-opus-4-6`; all runs executed on 2026-06-06. Haiku 4.5 is a dated snapshot, while Sonnet 4.6 and Opus 4.6 are versioned aliases, so the run date fixes which model each resolved to.

Table 3: Compilation reliability (SQ3): the percentage of compilations that loaded and passed the overlay's worked examples, by model and temperature.

Model	T=0.0	T=0.3	T=0.7
Fixed reference overlays (40 compilations per cell)			
Haiku 4.5	100%	100%	100%
Sonnet 4.6	100%	100%	100%
Opus 4.6	100%	100%	100%
Model-evolved overlays (240 compilations per cell)			
Haiku 4.5	88%	89%	88%
Sonnet 4.6	84%	85%	86%
Opus 4.6	86%	89%	87%

6.3 Results

We report three results in turn, each building on the last. We first establish that compilation is reliable and that sampling temperature does not affect it, which makes behaviour the informative question rather than whether code is produced at all. We then measure behavioural reproduction against the hand-written references, and finally convergence between independent compilations of the protocols the models evolve themselves. The two measures are read at the functional and representational levels defined in Section 6.2.

Compilation and temperature

A first result is negative and useful: sampling temperature does not affect the outcome. Compilation is reliable at every setting, with all 360 compilations of the four fixed reference protocols loading and passing their own worked examples (Table 3) and the harder evolved protocols compiling 84% to 89% of the time; both rates, and functional agreement, hold flat across 0.0, 0.3, and 0.7, and the representational rate shows no trend across them. We therefore pool temperatures in the behavioural tables, giving 30 compilations per cell of Table 4; this pooling follows from the flat rates, not an assumption. These are genuinely different programs: depending on protocol and temperature, between 3% and 70% of compilation pairs of a single document have byte-distinct source, so the high agreement reflects convergent behaviour rather than identical (memorised) code.

Behavioural reproduction

Table 4 reports reproduction at the two levels. At the *functional* level, agreement is complete: every compilation reproduces the reference's documented behaviour, for all four protocols and all three models, including the ordering, deduplication, and hash-verification the prose prescribes. This rate is conditional on the acceptance gate: only compilations that pass their worked examples are scored. It therefore measures whether those compilations also agree on behaviour beyond the examples, which they do. At the *representational* level, agreement is partial and uneven: only echo, whose observable state is a flat list of strings rather than a record, reproduces the reference exactly, while the three record-shaped protocols diverge by a margin that depends more on which model wrote the code than on the protocol. Independent compilations attach different metadata to the records they keep and name internal fields differently, so code that reproduces the documented behaviour can still store different bytes. Because this state never crosses the wire, the divergence does not affect interoperation.

Table 4: Behavioural reproduction (SQ3): per-cell percentage of 30 compilations that reproduce the hand-written reference, by model (H Haiku 4.5, S Sonnet 4.6, O Opus 4.6) and level.

	Functional			Representational		
Protocol	H	S	O	H	S	O
Echo	100%	100%	100%	100%	100%	100%
Content	100%	100%	100%	0%	40%	0%
Payment	100%	100%	100%	10%	60%	0%
File transfer	100%	100%	100%	0%	23%	70%

Table 5: Convergence (SQ4): per-protocol percentage of model-evolved compilation pairs that reach the same observable state, for same-model and cross-model pairs at each level.

	Functional		Representational	
Protocol	same model	different model	same model	different model
Echo	96%	84%	57%	41%
Content	86%	76%	82%	74%
Payment	92%	92%	49%	49%
File transfer	79%	24%	61%	18%

Convergence

For a fixed reference, wire-level interoperability is near-complete by construction, so the informative test is on the evolved protocols, where the specification itself varies (Table 5). The headline is functional convergence within a model: two compilations a single model makes of a protocol it evolved reach the same behaviour 79% to 96% of the time across all four protocols, the operative case for a community that standardises on one model. Representational convergence is lower and uneven, as in the reproduction results, and likewise never crosses the wire. Convergence is reliable within a model. Across different models it stays close to the same-model level for echo, content, and payment, with payment showing no penalty at all. It collapses only on the most complex protocol, file transfer, where functional agreement drops from 79% within a model to 24% across models.

6.4 Threats to Validity

Four caveats bound the study. For *construct* validity, the reference implementations and adversarial scenarios are author-written, so they probe behaviours we anticipated; an independent set would strengthen the metric (Section 9). The adversarial battery stresses handler inputs, not the language subset itself, so the containment of Section 5.2 is enforced by construction rather than tested against escape attempts. For *internal* validity, we vary only the compilation randomness, with the pipeline, encoding, language subset, and scenarios fixed

and fingerprinted each trial. *External* validity is bounded by three models of one family, four protocols, and three temperatures, with other vendors' models and cryptographic-ratchet protocols [33] out of scope. For *conclusion* validity, each rate is an exact proportion over all compilations in its cell, not an estimate from a subsample, so the rates carry no sampling error; the 100% functional rates and the spread in the representational rates are properties of the measured set itself.

7 Responsible Research

7.1 Reproducibility

The channel, its compiler, the experiment code, and the four protocol documents are in a public repository.[2] Dependency and language versions are pinned, and the per-run records and the per-cell aggregates behind Tables 4 and 5 are released alongside the code. With an Anthropic API key the whole study reruns from a single command; without one, the same code can be pointed at any compatible model endpoint, at the cost of evaluating a different model family. The method reproduces exactly. The reported numbers reproduce as far as the two aliased models (Section 6.2) are unchanged since the run date; the dated Haiku snapshot reproduces exactly.

The coverage matrix of Section 4 was produced by a single author. To limit this subjectivity, we anchor every cell to exact source quotations, released as a per-property source map with the code, that any reader can re-check; an independent second coding remains future work (Section 9).

7.2 Ethics

The study involves no human subjects and no live attacks on third-party systems. The only outside services the channel touches are the Anthropic interface for the language models and a test payment network used for admission; the test network carries no real money, and its free faucet caps any autonomous spending. That same free faucet means admission is costless, so the prototype's admission deters duplicate identities only nominally; Sybil resistance is out of scope and left to future work (Section 9). The properties of Section 3 describe structural gaps in published protocols, not vulnerabilities in deployed systems, so no coordinated-disclosure process applies.

7.3 Use of Large Language Models

Claude models were used in three roles, kept separate here so each can be judged. As a *research and prototyping collaborator*, Opus 4.7 helped design the schema, write the compiler and experiment code, and draft this paper; every code change we kept passes the project's test suite, and every paragraph was checked against the project's recorded research criteria. As the *agent runtime* of the deployed channel, Haiku 4.5 backs the autonomous agents that exchange messages, so it is part of the trusted computing base (Section 2). As the *channel under test*, the compiler of Section 5.1 prompts a Claude model, and the study evaluates the same family across three tiers (Haiku 4.5, Sonnet 4.6, Opus 4.6). Evaluating a Claude-compiled channel with Claude models is a deliberate scoping choice; its limit, that the findings speak to this model family and not others, is stated among the threats (Section 6.4).

[2] https://github.com/stubbaTU/DelftClaw/tree/communication-protocol

8 Discussion

At the wire level, independent compilations interoperate by construction: the document's encoding tables fix the byte layout. The informative question is behavioural, and there the result is clean. Every compilation reproduces the reference's documented behaviour, for all four protocols and three models. The only divergence is one level down, in the metadata each compilation attaches to the records it stores, a known effect of non-deterministic code generation [18]. This state never crosses the wire, so it does not affect interoperation; it would matter only for a feature that copies state between agents, like the shared signed log. The looseness is in the document, not in the models, which follow the prose faithfully. A schema that pins each record's shape as firmly as the wire format should close the gap, but we have not tested that: it is an open question, not a fix we claim.

Convergence, where the models evolve the protocols themselves, tells a sharper story. Two compilations that share a model reach the same behaviour for all four protocols, the case for a community on one model. Across different models the agreement holds for every protocol but the most complex, where it falls away (Section 6). A single-model community is safe; a mixed-model one should expect trouble on the hardest protocols. This bounds a useful finding: at the functional level the cheapest tier reproduces the reference as reliably as the most expensive, and it is the model that already runs the deployed agents (Section 7). Such a community needs no frontier model and can run on its members' own machines.

Together, the channel delivers self-verifying specifications under realistic compilation (P1), and with compile-time containment (P2) and custodian-free admission (P3) it provides the three agent-specific properties that no protocol in Table 1 offers together. Two limits bound this. The four protocols are a ladder of difficulty, not a complete sample; a wider set, with more record-shaped protocols, would map the representational gap more finely. And several agent-to-agent needs fall outside the channel altogether (Section 9).

One scope note matters. The decentralisation claim is about the channel, not the demo's bootstrap. The channel has no central authority: protocols and the network manifest are named by their content, transport is peer-to-peer over IPv8, and admission is settled by every peer replaying the signed donation logs, with no gatekeeper holding the keys. What the harness centralises is only discovery: it builds one manifest and injects it into every agent, so a run reproduces on one host. This is a normal out-of-band step in any peer-to-peer system. In a real deployment the founder writes the manifest once and a joiner receives it through a side channel, with content-addressing taking the orchestrator's place once the naming hash is collision-resistant (Section 5.1).

9 Conclusion and Future Work

We asked: *To what extent can autonomous LLM agents implement and evolve a shared communication protocol when*

each agent independently compiles it from a transmitted description, without central infrastructure, and what channel design enables this?

Existing protocols are not enough. The five properties of Section 3 all pass the agent-specificity test, and three are new to autonomous agents; Table 1 shows that none of the six existing protocols provides that triple $\{P1, P2, P3\}$ together. DelftClaw provides all three: each implementation checks itself against the test vectors its document carries, the compiled code is contained before it runs, and the community decides who belongs by replaying a shared signed log with no custodian.

On the precondition that independent compilations behave alike, the answer is positive, with one exception. Compilation almost always succeeds, wire agreement is near-perfect, and every compilation reproduces its hand-written reference at the functional level, with no loss as the protocols grow more complex. The only divergence is the stored metadata, which a tighter document can remove. When the models evolve the protocols themselves, two compilations that share a model converge for all four protocols; across different models this holds for all but the most complex (Section 6).

Several directions follow. To strengthen the evidence, an independent rater's test vectors would test P1 without the author's own examples; other model families would test how far the results carry beyond Claude; and red-teaming the contained code would test P2, which we argued for but did not attack. Whether a tighter schema closes the representational gap is the experiment our results most invite. To extend the channel, delegation provenance [29, 34], goal attestation, and per-principal rate limits ride over any transport, and Sybil-resistant admission is needed because ours is costless. The central open question is how convergence behaves at scale: we measured it on two or three nodes, so a whole population is the next step.

The wider aim is protocol emergence: a population can change its protocol by sending a description each agent compiles, so protocols can evolve at the speed of communication rather than through coordinated redeployment, turning a multi-year migration into a message. The reliability we measured is the precondition; carrying it to protocols that emerge from agent interaction, and measuring how reliably they converge across a population, is the direction we find most compelling.

References

[1] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460, IETF, 1998. Obsoleted by RFC 8200.

[2] J. Postel. Internet protocol. Technical Report RFC 791, IETF, 1981.

[3] Google. IPv6 adoption statistics. https://www.google.com/intl/en/ipv6/statistics.html, 2026. Accessed June 2026.

[4] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, IL, 1966. Edited and completed by Arthur W. Burks.

[5] Nikola Emilov. DelftClaw: Source code and experimental artifacts. https://github.com/stubbaTU/DelftClaw/tree/communication-protocol, 2026.

[6] Peter Steinberger et al. OpenClaw: An open-source personal AI assistant. https://openclaw.ai/, 2025.

[7] Yingxuan Yang, Huacan Chai, Yuanyi Song, Siyuan Qi, Muning Wen, Ning Li, Junwei Liao, Haoyi Hu, Jianghao Lin, Gaowei Chang, Weiwen Liu, Ying Wen, Yong Yu, and Weinan Zhang. A survey of AI agent protocols. arXiv:2504.16736, 2025.

[8] Abul Ehtesham, Aditi Singh, Gaurav Kumar Gupta, and Saket Kumar. A survey of agent interoperability protocols: MCP, ACP, A2A, ANP. arXiv:2505.02279, 2025.

[9] Xin Li, Mengbing Liu, and Chau Yuen. LLM agent communication protocol (LACP) requires urgent standardization: A telecom-inspired protocol is necessary. arXiv:2510.13821; NeurIPS 2025 AI4NextG Workshop, 2025.

[10] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon. The Messaging Layer Security (MLS) protocol. RFC 9420, IETF, 2023.

[11] B. Beurdouche, E. Rescorla, E. Omara, S. Inguva, and A. Duric. The Messaging Layer Security (MLS) architecture. RFC 9750, IETF, 2025.

[12] Decentralized Identity Foundation. DIDComm messaging specification v2.1. https://identity.foundation/didcomm-messaging/spec/v2.1/, 2024.

[13] A2A Project. Agent2agent (A2A) protocol specification. Linux Foundation; https://a2a-protocol.org/latest/specification/, 2025. Originally developed by Google.

[14] IBM Research and the BeeAI Community. Agent communication protocol (ACP). Linux Foundation; https://github.com/i-am-bee/acp, 2025.

[15] Samuele Marro, Emanuele La Malfa, Jesse Wright, Guohao Li, Nigel Shadbolt, Michael Wooldridge, and Philip Torr. A scalable communication protocol for networks of Large Language Models (Agora). arXiv:2410.11905, 2024.

[16] Gaowei Chang et al. Agent Network Protocol technical white paper. arXiv:2508.00007, 2025.

[17] Tribler Research Group. py-ipv8: The IPv8 networking layer. https://github.com/Tribler/py-ipv8, 2024.

[18] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of ChatGPT in code generation. arXiv:2308.02828, 2023.

[19] Christian Schroeder de Witt et al. Open challenges in multi-agent security: Towards secure systems of interacting AI agents. arXiv:2505.02077, 2025.

[20] Sahar Abdelnabi, Kai Greshake, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on*

Artificial Intelligence and Security (AISec '23). ACM, 2023.

[21] Yedidel Louck, Ariel Stulman, and Amit Dvir. Security analysis of agentic AI communication protocols: A comparative evaluation. *ACM Transactions on AI Security and Privacy*, 2025.

[22] Dezhang Kong, Shi Lin, Zhenhua Xu, Zhebo Wang, Minghao Li, Yufeng Li, Yilun Zhang, Hujin Peng, Xiang Chen, Zeyang Sha, Yuyuan Li, Changting Lin, Xun Wang, Xuan Liu, Ningyu Zhang, Chaochao Chen, Chunming Wu, Muhammad Khurram Khan, and Meng Han. A survey of LLM-driven AI agent communication: Protocols, security risks, and defense countermeasures. arXiv:2506.19676, 2025.

[23] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, 2003.

[24] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. RFC 8446, IETF, 2018.

[25] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. arXiv:2211.09527, 2022.

[26] Christopher Allen. The path to self-sovereign identity. Online essay, https://github.com/WebOfTrustInfo/self-sovereign-identity, 2016.

[27] Quinten Stokkink and Johan Pouwelse. Deployment of a blockchain-based self-sovereign identity. In *2018 IEEE International Conference on Internet of Things (iThings)*, 2018.

[28] Quinten Stokkink, Georgy Ishmaev, Dick Epema, and Johan Pouwelse. A truly self-sovereign identity system. In *IEEE 46th Conference on Local Computer Networks (LCN)*, 2021.

[29] Sandro Rodriguez Garzon, Awid Vaziry, Enis Mert Kuzu, Dennis Enrique Gehrmann, Buse Varkan, Alexander Gaballa, and Axel Küpper. AI agents with decentralized identifiers and verifiable credentials. In *Proceedings of the 18th International Conference on Agents and Artificial Intelligence (ICAART)*. SciTePress, 2026.

[30] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65. Springer, 2002.

[31] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and CRL profile. RFC 5280, IETF, 2008.

[32] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *Advances in Cryptology – CRYPTO 2017*, pages 570–596. Springer, 2017.

[33] T. Perrin and M. Marlinspike. The Double Ratchet algorithm. Signal specification, https://signal.org/docs/specifications/doubleratchet/, 2016.

[34] I. Herman, M. Sporny, D. Longley, O. Steele, and D. Reed. Verifiable credentials data model v2.0. W3C Recommendation, https://www.w3.org/TR/vc-data-model-2.0/, 2025.

A Echo Overlay Document

The `echo` reference overlay (Section 6.1) is reproduced in full below as an example of a *markdown-as-overlay* document: the Markdown a model compiles into a running IPv8 peer group. It is the simplest of the four protocols the study compiles, a stateless round trip whose only observable state is the list of received replies. It shows the section structure every overlay follows: an identity block whose hash names the community, message and field tables that pin the wire format, with each message's handler given in prose, the runtime state, and the test vectors the compiled code must reproduce before use.

```
# Identity

- name: echo
- version: 1.0.0
- description: Round-trip echo overlay for compiler testing.
- lifecycle: peer-observer

# Messages

## ECHO_REQUEST

- msg_id: 1

| name | encoding | description |
|------|----------|-------------|
| payload | varlenH-utf8 | utf-8 string the receiver will echo
    back |

### Handler

On receipt of ECHO_REQUEST, send ECHO_RESPONSE whose payload is
    the
received payload (utf-8 decoded) with an exclamation mark
    appended,
re-encoded as utf-8 bytes.

## ECHO_RESPONSE

- msg_id: 2

| name | encoding | description |
|------|----------|-------------|
| payload | varlenH-utf8 | utf-8 string the sender originally
    requested, with a trailing exclamation mark |

### Handler

On receipt of ECHO_RESPONSE, append the decoded utf-8 string to
``self.received_responses``. The sender side correlates responses
out-of-band (tests poll ``received_responses`` directly).

# Runtime State

| name | type | description |
|------|------|-------------|
| received_responses | list[str] | utf-8 strings collected from
    ECHO_RESPONSE messages, in arrival order. Tests poll this
    directly. |

# Errors

| code | name | policy |
|------|------|--------|
| 1 | malformed_payload | drop |

# Dependencies
```

```
(none)

# Test Vectors

## ECHO_REQUEST

- fields: {payload: }
  bytes: 0000

- fields: {payload: hi}
  bytes: 00026869

## ECHO_RESPONSE

- fields: {payload: }
  bytes: 0000

- fields: {payload: hi!}
  bytes: 0003686921
```