SlimWire Protocol (SWP): A Binary Framing Substrate for Federated AI Agent Interoperability

Jericko Tejido

February 20, 2026

Abstract

The rapid emergence of LLM-based “agent” ecosystems has produced multiple application-level protocols for tool invocation and agent collaboration, notably the Model Context Protocol (MCP) and Agent2Agent (A2A), without a shared, minimal, independently implementable wire substrate beneath them. MCP focuses on tool semantics using JSON-RPC over transports such as stdio and HTTP streaming [1] , while A2A focuses on agent-to-agent task delegation and discovery [2]. As deployments move into federated, cross-boundary settings, the lack of a common record/framing layer, a mandatory-to-implement encoding, explicit security binding requirements, and machine-verifiable conformance artefacts increases integration friction and makes interoperability claims hard to validate. This technical report presents SlimWire and its wire profile family, referred to here as the SlimWire Wire Protocol (SWP): a slim binary framing and envelope substrate with (i) length-prefixed stream framing, (ii) a compact mandatory envelope encoding (E1) using unsigned LEB128-style varints and a TLV extension block, (iii) profile dispatch by stable numeric profile identifiers, (iv) a mandatory payload encoding binding (P1) using Protocol Buffers for non-MCP profiles [3], and (v) a conformance model comprising conformance classes and machine-readable golden vectors executed by a spec-vector runner with reproducible artefact bundles.

Executive summary

Contemporary LLM systems increasingly behave as *distributed systems*: they invoke external tools, delegate long-running tasks to other agents, stream intermediate progress, exchange artefacts (files/structured outputs), and operate across organisational trust boundaries. In practice, “agent communication” spans heterogeneous interaction shapes: tool invocation (agent → tool server), delegation (agent → agent), event streaming (progress/log/audit), observability propagation (trace correlation), and durable delivery or relay (store-and-forward). Today, these behaviours are commonly implemented as application protocols over HTTP/JSON with a mixture of bespoke gateways, transport-specific streaming conventions, and deployment-specific security assumptions.

Two application-level protocol efforts have gained traction. MCP standardises tool invocation and discovery as JSON-RPC with defined transports such as stdio and “Streamable HTTP” [1]. A2A standardises agent-to-agent interoperability, including capability discovery via an “Agent Card” and task lifecycle semantics [2]. Both are valuable; neither primarily aims to be a transport-independent, binary framing substrate with a conformance regime designed for independent implementations. The consequence is that cross-protocol and cross-organisation composition remains largely an integration exercise rather than a verifiable engineering contract: message boundary detection, size limiting, replay policy, correlation, identity surfaces, and failure semantics are frequently reimplemented at every gateway or “adapter” boundary.

SlimWire/SWP addresses this by separating concerns into a thin Core substrate and evolvable profiles. SWP Core is a small record/envelope layer that answers *where messages begin and end* and *how to route them* without parsing application payloads. Above the Core, profiles define the semantics of tool invocation mapping, agent task lifecycle, and common infrastructure primitives required in federated deployments. The result is a protocol family that can carry multiple application protocols in a single connection with consistent validation, correlation, and security hooks.

SWP’s Core contribution is intentionally minimal:

- **Binary framing (record layer)**: each message is a frame on a byte stream encoded as a 32-bit big-endian length prefix followed by that many bytes of an encoded envelope. This is the simplest widely-used framing pattern in systems protocols and enables early rejection of oversized messages before any payload parsing.
- **A compact envelope**: seven required fields (Core version, profile identifier, message type, flags, sender timestamp, message identifier, and opaque payload bytes) provide just enough information for dispatch, correlation, and enforcement.
- **Mandatory encoding E1**: a compact, independently implementable envelope encoding in which envelope fields are encoded in a fixed order using unsigned LEB128 (“protobuf-style”) varints and length-delimited byte strings, plus a TLV extension block for forward compatibility.

- **Profile dispatch by numeric profile identifiers**: Core does not interpret payloads; it routes solely based on profile identifiers (`profile_id`) and profile-specific message types (`msg_type`). This enables relays and gateways that can forward messages without understanding application semantics.
- **Deterministic rejection/error surface**: Core establishes a bounded set of failure conditions at the framing/envelope boundary. Profiles define their own payload-level errors, but Core ensures parsers can fail fast and consistently.

Above the Core, SWP defines a profile family that deliberately bridges existing applications rather than replacing them:

- **MCP mapping profile** carries MCP JSON-RPC payloads as opaque bytes to minimise semantic drift and allow byte-preserving relays. MCP's normative JSON-RPC requirements are preserved (https://modelcontextprotocol.io/specification/).
- **A2A profile** represents A2A task lifecycle and discovery concepts in the profile layer while preserving A2A's application intent [2].
- **Extended infrastructure profiles** cover discovery, generic RPC with streaming and cancellation, structured events, artifact transfer, state exchange, credential/delegation propagation, policy hint propagation, observability correlation aligned with W3C Trace Context [6], and store-and-forward relay.

A core motivation for SWP is *verifiability*. SWP includes:

- **Conformance classes (C0–C5)**: a taxonomy that lets implementers make scoped claims (e.g., "SWP C1" meaning Core + MCP mapping) rather than ambiguous "we implement SWP" statements.
- **Golden conformance vectors**: machine-readable test cases comprising binary fixtures and expected results for both accept and reject paths.
- **Spec-vector runner and artefact bundles**: an executable runner produces reproducible JSON artefacts (with a versioned schema, run provenance, and per-vector "used_fallback" indicators), and supports strict implementation-only mode that fails any vector requiring fallback evaluation.

The practical outcome is a substrate that can be adopted incrementally. A deployment can start with Core + MCP mapping (C1) to reduce bespoke protocol bridging and establish consistent enforcement and correlation, then later adopt additional profiles as needed, without changing the Core substrate.

Motivation and problem statement

The central problem is not that agents cannot communicate, but that *heterogeneous interoperability is expensive and difficult to verify*. In a federated setting, a single "end-to-end action" often spans multiple protocol domains and trust boundaries. Without a shared substrate layer, each boundary becomes a bespoke engineering problem and a potential security and reliability hazard.

Concrete examples illustrate the gap.

Tool invocation with streaming progress and bounded resources. An orchestrator agent invokes multiple tools, some local and some remote, and requires progress updates for long-running operations. MCP provides a clear tool invocation model and standardised transports [1]. However, deployments still need consistent message correlation across hops, deterministic behaviour under malformed tool output, and an enforcement boundary that can reject oversized frames before parsing JSON. When MCP messages are proxied through gateways (e.g., to cross network boundaries or to unify transport mechanisms), each gateway typically reimplements size limits, JSON parsing, streaming semantics, and error reporting conventions.

Cross-organisation delegation with artefacts. Organisation A delegates a task to an agent in organisation B (A2A), B invokes tools via MCP, and B returns an artefact (file or structured output) along with progress events and trace correlation. A2A provides a discovery and delegation framework, including capability signalling [2], but it does not define a shared record layer that unifies message framing, payload limits, correlation, trace propagation, and conformance. Federation requirements—peer authentication, stable identity surfacing for authorisation/audit, replay resistance, and policy constraints—are often handled out-of-band with inconsistent semantics across system boundaries.

These examples surface three substrate-layer deficits that recur in real deployments:

1. **Message boundary detection and enforcement are reimplemented.** On raw byte streams, message boundaries do not exist unless explicit framing is defined. Mature protocols often standardise a record layer (e.g., HTTP/2 frame layer in RFC 9113 [4]) or a message framing convention (e.g., gRPC message framing over HTTP/2 [3]). Agent/tool ecosystems often leave this boundary to deployment or transport conventions.

2. **Security contracts are implicit.** Many deployments rely on TLS, but protocol profiles still need explicit assumptions: "is this connection authenticated?", "what is the peer identity?", "how is replay handled?", "must failures be fail-closed?". TLS 1.3 defines a secure channel and record layer (RFC 8446 [5]), but application protocols must still define how those properties are required and consumed.

3. **Conformance is not independently verifiable.** Without a conformance class taxonomy and deterministic golden vectors, “compatible” implementations are difficult to reason about. Mature protocol ecosystems operationalise interoperability through test vectors, conformance suites, and strict requirements language (RFC 2119 and RFC 8174). In the AI communications domain, this discipline is still emerging.

SWP’s scope is therefore deliberately narrow: it standardises the record/envelope substrate and the mechanisms by which profiles are identified, validated, and conformance-tested. Profiles standardise semantics above the substrate. SWP is designed to *carry* MCP and A2A rather than displace them.

Related work

SWP combines well-established ideas from systems protocol engineering with the emerging requirements of agent/tool interoperability. Most novelty lies in the *packaging* and *rigour* of these ideas for the AI communications context.

MCP and A2A

MCP defines a JSON-RPC-based application protocol for tool invocation and discovery, with transports including stdio and HTTP streaming [1]. MCP is effective for tool ecosystems because it prioritises accessibility and leverages existing HTTP infrastructure, but it does not define a transport-independent binary substrate or a substrate-level conformance class model.

A2A is positioned as an open standard for agent-to-agent interoperability, including capability discovery (“Agent Card”) and task lifecycle semantics [2]. Like MCP, A2A is primarily application-level and commonly realised over HTTP/JSON. A2A and MCP are thus best understood as *profile candidates* carried over a stable substrate rather than direct alternatives to a record layer.

gRPC and HTTP/2

HTTP/2 defines a binary framing layer (RFC 9113, https://datatracker.ietf.org/doc/html/rfc9113) that multiplexes streams and frames over a single connection. gRPC defines how RPC payload messages are framed and mapped onto HTTP/2 data frames, including message boundaries and optional compression indicators [3]. These systems demonstrate two relevant principles: explicit binary framing simplifies robust parsing and enforcement, and layering multiple framing conventions is normal (e.g., application message framing within HTTP/2 streams).

SWP differs in that its Core is transport-independent and profile-dispatch oriented, rather than tied to a single RPC method namespace. SWP can run over TCP, over TLS-protected streams, or over other transports that satisfy the S1 security binding.

Earlier agent communication standards: KQML and FIPA

Agent interoperability is not new. KQML (Knowledge Query and Manipulation Language) standardised agent communication acts and message structures for distributed AI systems, with public documentation hosted by UMBC [8]. FIPA produced specifications for agent communication languages and message representations (e.g., FIPA ACL representation specification [9]). These efforts focused largely on *semantic intent* and communicative acts rather than on a modern system substrate with binary record framing, federated security bindings, and machine-verifiable conformance vectors. SWP's emphasis is the infrastructure layer beneath semantics.

Comparison with MCP and A2A

The table below summarises the distinctions most relevant to systems deployment.

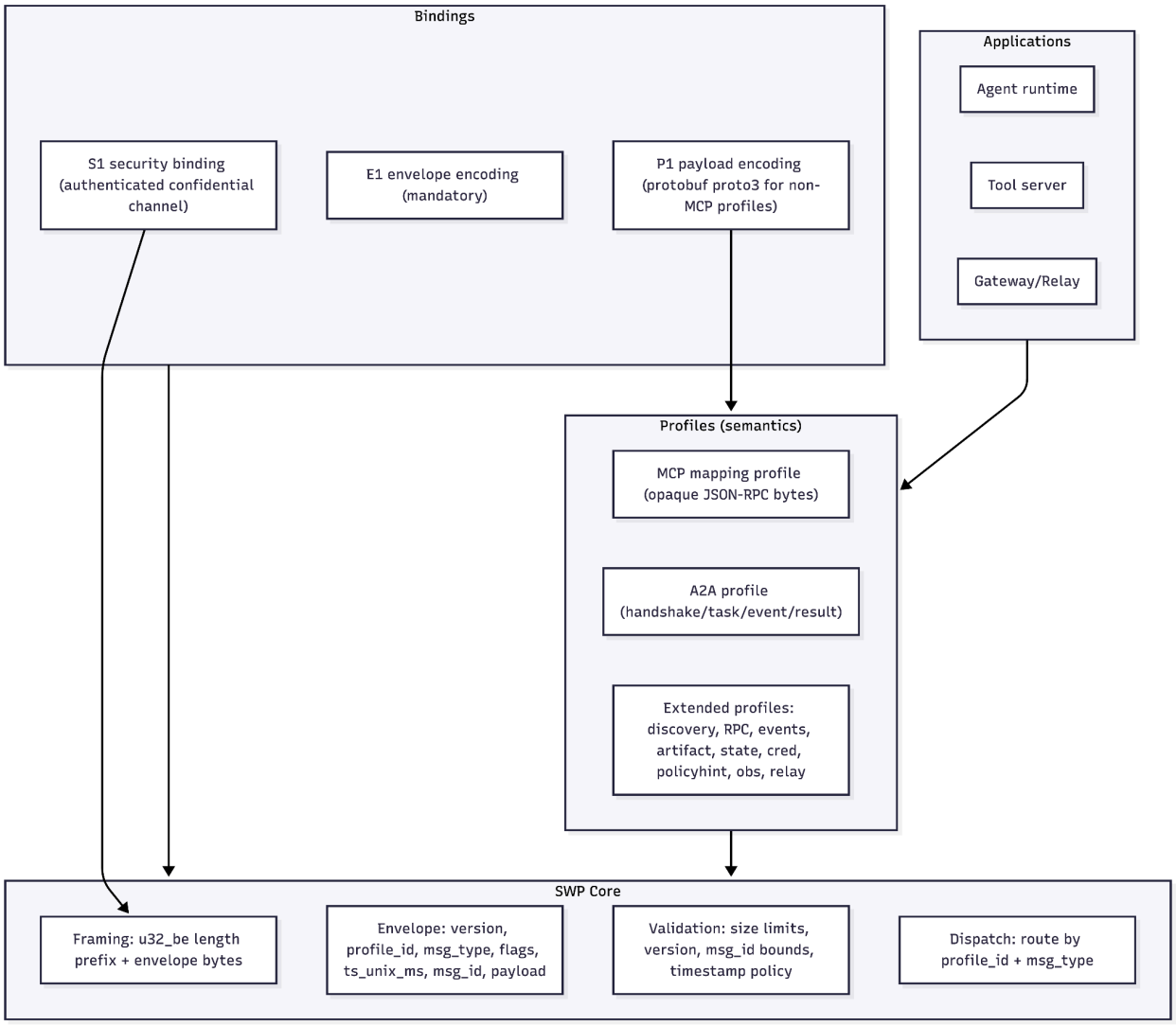
Dimension	SWP	MCP	A2A
Primary scope	Binary framing + envelope substrate with profile dispatch and conformance regime	Tool invocation and discovery semantics	Agent-to-agent delegation and discovery semantics
Framing / record layer	Length-prefix framing on byte streams: `u32_be N` + `N` bytes envelope	Transport-defined (e.g., HTTP streaming, stdio)	Transport-defined (commonly HTTP/JSON)
Mandatory wire encoding	Yes: E1 mandatory envelope encoding	No substrate encoding (JSON-RPC structures)	No substrate encoding (application JSON)
Payload encoding	P1 protobuf for non-MCP profiles; MCP mapping preserves	JSON-RPC (UTF-8), per spec	JSON documents/messages, per spec

Dimension	SWP JSON-RPC bytes	MCP	A2A
Profile/extension model	Numeric `profile_id` registry + per-profile `msg_type`	Methods and schemas within JSON-RPC	A2A concepts/extensions; discovery document
Federation/security story	Security binding S1 specifies channel properties; profiles can assume stable peer identity and fail-closed behaviour	Largely deployment-defined	Largely deployment-defined; discovery signals auth requirements
Conformance model	Conformance classes (C0–C5) + golden vectors + artefact bundles + strict mode	Spec exists; conformance artefacts not defined as a substrate suite	Spec exists; conformance artefacts not defined as a substrate suite
Intended interop mode	Bridge/relay substrate beneath MCP and A2A plus additional infrastructure profiles	Direct tool client–server	Direct agent–agent interoperability

Design of SlimWire/SWP

SWP is a layered protocol family. The Core standardises *framing, envelope, dispatch, and minimal validation*; everything else is in profiles and bindings. This separation reduces the blast radius of fast-changing application semantics.

Architecture overview



Core framing and envelope

Stream framing (`u32_be` length prefix). A SWP stream is a sequence of frames. Each frame is encoded as:

- a 32-bit unsigned integer `N` in network byte order (big-endian), followed by
- exactly `N` bytes of an encoded envelope.

Receivers reject frames with truncated prefixes, `N = 0`, `N` exceeding `MAX_FRAME_BYTES`, truncated bodies, or envelopes that cannot be decoded by the chosen envelope codec. This length-prefix design enables early size enforcement before payload parsing; it follows the common record-layer approach used in many binary protocols and is structurally similar to gRPC's explicit message framing within HTTP/2 streams [3].

Envelope fields. The Core envelope carries seven required fields:

- `version` (uvarint): Core protocol version
- `profile_id` (uvarint): profile dispatch identifier
- `msg_type` (uvarint): message type within the profile
- `flags` (uvarint): core behaviour flags (unknown bits must not change interpretation)
- `ts_unix_ms` (uvarint): sender timestamp in Unix milliseconds (policy-controlled freshness)
- `msg_id` (bytes): message correlation identifier (bounded length)
- `payload` (bytes): opaque profile payload

The Core does not interpret payload bytes. This allows relays/gateways to forward traffic without parsing application semantics and makes the Core stable under profile evolution.

E1 envelope encoding

E1 is the mandatory-to-implement envelope encoding binding for Core v1. E1 encodes fields in a fixed order using two primitives:

- **Unsigned varints (`uvarint`) encoded as LEB128**: lower 7 bits per octet, MSB as continuation, least-significant group first. This is the same family of varint encoding used by protobuf and many other systems protocols.
- **Length-delimited byte strings (`bytes`) encoded as** `uvarint length` followed by `length` octets.

E1 encodes the envelope (within the `N` envelope bytes) in this exact order:

1. `version`: `uvarint` (must be `1` for Core v1)
2. `profile_id`: `uvarint`
3. `msg_type`: `uvarint`
4. `flags`: `uvarint`
5. `ts_unix_ms`: `uvarint`
6. `msg_id`: `bytes`
7. `extensions`: `bytes` (TLV block; may be empty)
8. `payload`: `bytes`

Bounded decoding rules. To avoid common parser exploitation classes, E1 parsers reject malformed or unbounded varints. Implementations reject varints longer than 10 octets or those that overflow 64-bit unsigned range. This is consistent with defensive parsing guidance common in varint-based protocols.

TLV extension block. The `extensions` field contains a sequence of TLV entries:

- `ext_type`: `uvarint`
- `ext_val`: `bytes`

Unknown `ext_type` values are ignored. Extension type ranges are reserved to prevent collisions (e.g., low values reserved for Core/bindings; higher values for profile-defined extensions). The extension block exists to carry forward-compatible envelope-level metadata (e.g., additional correlation or routing hints) without requiring a Core version bump.

P1 payload encoding binding

Profiles define semantics and payload schemas. SWP mandates the payload encoding binding **P1** for all profiles except the MCP mapping profile. P1 specifies **Protocol Buffers (proto3) wire format** as mandatory-to-implement for relevant profiles[10].

The rationale is systems-oriented:

- **Byte-level conformance testing.** Protobuf's binary encoding supports deterministic fixtures, enabling golden vectors that exercise payload decoding and validation rather than only envelope parsing.
- **Schema evolution discipline.** Field numbers must not be reused; removed fields must be reserved; consumers ignore unknown fields for forward compatibility[10].
- **Efficiency.** Compact payloads matter for high-volume event streams and artifact transfer metadata.

MCP mapping as the explicit exception. To preserve MCP semantics and avoid semantic drift, the MCP mapping profile carries MCP JSON-RPC payload bytes opaquely. This aligns with MCP's normative JSON-RPC framing expectations[1] and allows a relay to be byte-preserving without JSON parsing.

Profile registry and `profile_id` allocation

Core dispatch depends on a stable registry of numeric `profile_id` values. SWP adopts a non-reuse policy and documents reserved ranges.

A representative initial allocation (as of this draft) is:

- `1`: MCP mapping profile

- `2`: A2A profile
- `3–9`: reserved for future foundational profiles/bindings
- `10–19`: extended infrastructure profiles (discovery, RPC, events, artifact, state, credentials, policy hints, observability, relay)

Each profile defines its own `msg_type` values and payload schema (for P1 profiles, via normative `.proto` annexes). The registry is intended to evolve via governance rules (e.g., rationale required, conformance vectors added, compatibility impact stated).

Conformance model, vectors, and error taxonomy

Conformance classes (C0–C5). SWP defines a cumulative taxonomy that lets implementers claim scoped subsets. A C1 implementer need not implement the entire extended profile family; a consumer of the claim can understand exactly which namespaces must pass.

Golden vector suite. The conformance suite consists of machine-readable vectors: each vector pairs (i) a `.bin` file containing the exact bytes of a framed message (length prefix + E1 envelope bytes) and (ii) a `.json` descriptor containing expected outcome and assertions. Vectors cover positive cases, negative cases that exercise every rejection path, boundary cases at exact limits, and process/policy checks.

Spec-vector runner and strict mode. The runner executes vectors and produces reproducible JSON summaries. Two modes are supported:

- **Default mode** allows fallback evaluation where a vector exercises policy/state behaviours not yet implemented in a specific handler, while still validating decoding/validation invariants and expected outcomes.
- **Strict mode (`-no-fallback`)** fails any vector that requires fallback evaluation, providing an "implementation-only" signal for publication claims and external implementers.

Error codes: canonical vs alias. To support interoperability across implementations that may use different internal error taxonomies, SWP distinguishes a canonical conformance error taxonomy (e.g., `ERR_INVALID_FRAME`) from implementation/runtime alias codes (e.g., `INVALID_FRAME`). Conformance artefacts carry both where applicable so that independent implementations can map their internal codes to canonical codes while preserving local error reporting.

Security, implementation, and evaluation

Security considerations and S1 binding

SWP purposely does not hard-code a single transport security protocol. Instead it defines a **security binding S1**: a set of channel properties that must hold before any frame is processed. S1 requires:

- confidentiality (passive observers cannot read frames),
- integrity (active attackers cannot modify frames undetected),
- peer authentication (endpoints can identify the other side),
- replay resistance (via transport anti-replay and/or freshness policy),
- downgrade resistance (fail-closed negotiation).

TLS 1.3 provides a widely deployed instantiation of the required properties and defines the protected record layer for application data (RFC 8446[5]. In transport terms, S1 is compatible with multiple underlying channels, including TLS-over-TCP and (where relevant) QUIC, which uses TLS 1.3 semantics for security (RFC 9000 and RFC 9001, https://datatracker.ietf.org/doc/html/rfc9000 and https://datatracker.ietf.org/doc/html/rfc9001).

S1's operational consequences are deliberately strict:

- frames must not be processed until channel establishment completes,
- authentication/integrity failures terminate the channel (fail-closed),
- peer identity must be surfaced to profile handlers for authorisation and audit logging.

SWP includes a substrate-level timestamp field (`ts_unix_ms`) that deployments may use for freshness windows (e.g., rejecting frames outside `MAX_CLOCK_SKEW_MS`), but timestamp enforcement is policy-controlled because clock synchronisation differs across environments.

Credential/delegation and policy profiles

Cross-boundary systems commonly require "act on behalf of" semantics and constraint propagation. SWP includes a credential/delegation profile whose goal is *propagation*, not the definition of a new IAM system: credentials are carried as opaque tokens with

explicit expiry, optional chain semantics, and chain-length limits. Policy hints are carried as portable constraints (e.g., residency, cost budgets, network restrictions) with a defined conflict/violation signalling surface.

Observability alignment

Distributed tracing propagation is standardised by W3C Trace Context (`traceparent` and `tracestate` [6]). OpenTelemetry adopts W3C Trace Context as the default propagation format and documents context propagation behaviour [7]. SWP's observability profile aligns with these standards so that trace correlation can cross agent/tool boundaries without proprietary propagation semantics. A key security/privacy consideration is preventing sensitive identifiers from leaking into propagated context (especially "baggage"-style key/value mechanisms).

Implementation status

SWP is shipped as a spec kit with normative documents, proto annexes for P1 profiles, and an executable "proof of concept" (PoC) implementation used to generate and validate conformance vectors. The PoC is a reference artefact rather than a production system; its purpose is to exercise the framing/envelope codec, profile routing, and conformance harness end-to-end.

A practical issue in early protocol efforts is drift between a rapid PoC and the spec-track encoding. In SWP, the earlier mismatch—PoC handlers parsing JSON payloads while P1 specified protobuf—has been resolved, with P1 protobuf as the normative and implemented payload binding for non-MCP profiles. MCP mapping remains byte-preserving and JSON-RPC-oriented by design.

Evaluation plan, artefact bundles, and metrics

The primary evaluation mechanism is the conformance suite executed by the spec-vector runner. The runner emits a structured JSON summary with:

- a versioned schema identifier,
- run provenance (pattern, strict/no-fallback flag, timestamp, runner revision identifier),
- per-vector results (including `used_fallback`),
- canonical and alias error codes (where applicable).

This enables publication-quality reproducibility: a paper can cite both the command line used and the resulting JSON artefact bundle.

Artefact bundle commands (illustrative). The exact Make targets are repository-specific, but an SWP repository typically provides equivalents of:

```
# Default run (fallback evaluation permitted)
make vectors SPEC_VECTOR_ARGS="-pattern 'conformance/vectors/core_*.json'
-json-out artifacts/conformance/core.default.json"

# Strict run (implementation-only; fails if fallback used)
make vectors-strict SPEC_VECTOR_ARGS="-pattern
'conformance/vectors/core_*.json' -json-out
artifacts/conformance/core.strict.json"

# Convenience bundle targets (if provided by the repository)
make conformance-core
make conformance-all
```

A publication artefact bundle should include:

- the exact runner command lines,
- stdout logs from both default and strict runs,
- the JSON summaries,
- the runner output schema document,
- the conformance checklist and error code taxonomy documents.

Quantitative metrics for systems evaluation. Beyond “does it interoperate?”, SWP’s evaluation plan focuses on measurable systems properties:

- **Parser and codec overhead:** CPU cost and allocation behaviour of framing + E1 decoding under typical message sizes; rejection cost for malformed/oversized frames.
- **Wire overhead:** envelope size overhead under typical profiles (e.g., minimal frames; tool invocation frames; event frames).
- **Streaming behaviour:** throughput and latency for event streaming; backpressure behaviour at transport boundaries.
- **Adoption metrics:** number of independent implementations that pass the same conformance class suites (a common bar for mature protocol ecosystems), and number of real deployments that can reduce bespoke gateway code by standardising on Core + profiles.

Deployment scenarios, limitations, and non-goals

Interoperability and deployment scenarios

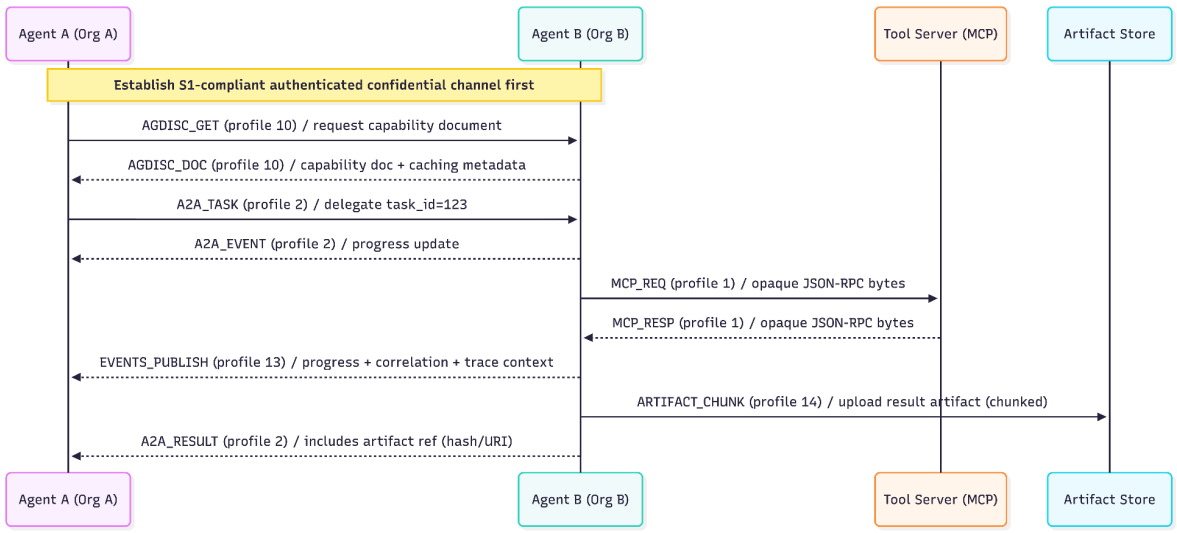
SWP is intended to be adopted incrementally. Three representative scenarios illustrate how it composes.

Scenario A: MCP bridging without semantic drift (C1). Deploy SWP Core + MCP mapping behind a gateway that translates between existing MCP transports (HTTP streaming or stdio) and SWP framing. MCP payload bytes remain opaque and preserved in relay mode, enabling byte-preserving federation without rewriting JSON-RPC messages.

Scenario B: Federated agent delegation (C2). Two organisations exchange agent tasks using SWP Core + A2A profile over an S1-compliant channel (e.g., mTLS). Peer identity surfaced by S1 is used for authorisation and auditing; task lifecycle invariants are evaluated at the profile layer.

Scenario C: Unified connection for RPC + events + artefacts (C3/C4/C5). A deployment extends to RPC, events, observability, and artifact transfer profiles so that a single secured connection carries requests, progress events, trace correlation, and artefact references or chunks. This avoids parallel “side channels” for observability and file transfer while using consistent correlation identifiers.

A sample federated workflow is shown below.



Limitations and non-goals

SWP is intentionally conservative in scope, and several items are explicit non-goals for the baseline:

- **Not a replacement for MCP or A2A.** SWP is designed to carry and bridge them.
- **Not a new IAM or policy language.** Credential and policy profiles propagate tokens and constraints; they do not define a full authorisation framework.
- **No message-level cryptography in the baseline.** S1 is a channel security binding. End-to-end encryption/signatures across relays are deferred to potential future bindings.
- **No Core-level multiplexing.** SWP Core is a record layer on a stream; multiplexing can be provided by transports (e.g., HTTP/2 [4]) or by higher-level profiles if needed.
- **Mermaid diagrams are documentation artefacts.** They are included as source blocks for reproducibility; arXiv PDF renderings typically require converting diagrams to figures during paper preparation.

Conclusion

SlimWire/SWP demonstrates that federated agent/tool interoperability admits a systems-style substrate solution: a slim record layer with a small envelope, a mandatory-to-implement encoding, stable profile dispatch, explicit channel security binding requirements, and a conformance model based on classes, golden vectors, and reproducible artefact bundles. The design borrows proven principles from adjacent protocol ecosystems—binary framing (HTTP/2), explicit message boundaries and schema discipline (gRPC/protobuf), and standards-style requirement language—while targeting an emerging AI communications space where fragmentation is still high. By carrying MCP and A2A as profiles rather than competing with them, SWP aims to reduce bespoke gateway logic and make interoperability claims verifiable.

Appendix

Conformance classes C0–C5

The following table summarises conformance classes as cumulative claims. A valid class claim requires 100% pass on required vector namespaces for that class.

Class	Intended claim	Required components (illustrative)
C0	Core baseline	Core framing + E1 encoding + S1 binding requirements
C1	MCP bridge	C0 + MCP mapping profile
C2	A2A baseline	C0 + A2A profile
C3	Runtime primitives	C0 + RPC + EVENTS + OBS profiles
C4	Data plane	C3 + ARTIFACT + STATE profiles
C5	Federation	C4 + AGDISC + TOOLDISC + CRED + POLICYHINT + RELAY profiles

Example conformance vector

This example illustrates the structure of a golden vector as used by the SWP conformance suite. It includes a binary fixture description (wire bytes) and the expected result descriptor (JSON). Field and file names are illustrative; repositories may differ in exact schema naming.

Vector ID: `e1_0001_valid_min_envelope`
Purpose: verify that a minimal E1-encoded envelope frame is accepted.

Binary fixture description (wire format):

- `frame_len`: `u32_be` length prefix of the envelope body (E1 bytes)
- `envelope_body` (E1 fields in order):
- `version = 1` (uvarint)
- `profile_id = 1` (uvarint)
- `msg_type = 1` (uvarint)
- `flags = 0` (uvarint)
- `ts_unix_ms = 0` (uvarint)

- `msg_id` = 16 bytes (length-delimited)
- `extensions` = empty (length-delimited bytes of length 0)
- `payload` = empty (length-delimited bytes of length 0)

An illustrative hex rendering (spacing for readability; values are example-only):

```
00 00 00 18 | 01 01 01 00 00 10 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 00 00
```

Where `00 00 00 18` is the big-endian envelope length (24 bytes in this example), and the remainder is the E1 envelope body.

Expected result (JSON descriptor excerpt):

```
{
  "vector_id": "e1_0001_valid_min_envelope",
  "expected": {
    "outcome": "accept",
    "assert": {
      "version": 1,
      "profile_id": 1,
      "msg_type": 1,
      "flags": 0,
      "ts_unix_ms": 0,
      "msg_id_len": 16,
      "payload_len": 0
    }
  }
}
```

Conformance artefact bundle guidance

For reproducibility in papers, interop reports, or CI, publish an artefact bundle containing:

- commands used (default and strict runs),
- runner stdout logs,
- JSON summary outputs,
- the runner output schema document,
- conformance class definitions and error code taxonomy.

Illustrative commands:

```
# Default (fallback permitted) and strict (no-fallback) runs for Core
make vectors SPEC_VECTOR_ARGS="-pattern 'conformance/vectors/core_*.json'
-json-out artifacts/conformance/core.default.json"
make vectors-strict SPEC_VECTOR_ARGS="-pattern
'conformance/vectors/core_*.json' -json-out
```

```
artifacts/conformance/core.strict.json"

# Optional: generate a complete bundle for the full suite if supported by the
repository
make conformance-all
```

Primary references cited inline in this report include:

1.) MCP specification (https://modelcontextprotocol.io/specification/)
2.) A2A specification and overview (https://a2a-protocol.org/; https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/)
3.) gRPC protocol documentation (https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md; https://grpc.io)
4.) HTTP/2 specification (RFC 9113, https://datatracker.ietf.org/doc/html/rfc9113)
5.) TLS 1.3 (RFC 8446, https://datatracker.ietf.org/doc/html/rfc8446)
6.) W3C Trace Context (https://www.w3.org/TR/trace-context/)
7.) OpenTelemetry context propagation (https://opentelemetry.io/docs/concepts/context-propagation/)
8.) KQML (https://www.cs.umbc.edu/research/kqml/)
9.) FIPA ACL representation (https://www.fipa.org/specs/fipa00071/XC00071B.pdf)
10.) Protocol Buffers proto3 guide (https://protobuf.dev/programming-guides/proto3/).