HAPPI/1.0

Protocol, Possibilities, and the Path to AI as Infrastructure

CodeTonight (Pty) Ltd t/a ENTER Konsult, Cape Town

MESH Council Synthesis · April 2026

Working Paper — Internal + Partner Distribution

Abstract. HAPPI (HAL API Protocol Interface) is an open wire-protocol specification for AI integration: a JSON envelope in, an NDJSON event stream out, seven defined event types. HAL is the name of a reference runtime for HAPPI currently in development and evaluation by the authors; any HAPPI-compliant runtime may substitute for it. This paper documents the full possibility space of HAPPI/1.0 as established by a twelve-agent MESH deliberation council — a multi-agent review pattern in which each agent analyses the problem from a distinct specialist perspective (protocol design, security, compliance, developer experience, ecosystem dynamics, and so on), and the outputs are synthesised into a consolidated finding. The use of multiple agents reduces single-model blind spots and surfaces a broader range of considerations than any individual analysis would produce. The paper is cross-referenced against the canonical whitepaper (`happi.md`). We cover multi-model orchestration patterns, polyglot `.md` workflow composition, agentic recursion via `sub_request`, custom domain harnesses, enterprise compliance implications, and the long-arc vision of AI reasoning as a Unix-level primitive. Where claims are prospective we state falsification conditions explicitly.

Contents

---

1 The Protocol — What HAPPI Actually Is

1.1 Core Design

In this paper, `hal -happi-api` appears in many code examples as the CLI invocation. It refers to the HAL runtime's command-line interface — the reference implementation of HAPPI, not a separate product. Any HAPPI-compliant runtime can stand in its place.

HAPPI is a request/response protocol operating at the wire level. Every call is one JSON envelope; every response is a stream of NDJSON events.

Input — the envelope:

```
{
  "v": "happi/1.0",
  "id": "req-001",
  "cmd": "anthropic.messages.create",
  "args": [{ "model": "claude-opus-4-7",
             "messages": [{ "role": "user",
                           "content": "Your question" }] }],
  "flags": { "provider": "anthropic/claude-opus-4-7",
             "max_tokens": 4096 },
  "auth":  { "scheme": "apikey",
             "token": "env:ANTHROPIC_API_KEY" }
}
```

Output — the event stream:

```
{"type":"started",   "request_id":"req-001"}
{"type":"delta",     "request_id":"req-001", "text":"The answer "}
{"type":"delta",     "request_id":"req-001", "text":"continues here..."}
{"type":"completed", "request_id":"req-001"}
```

The seven event types — and only seven:

Event	Meaning
`started`	Inference began
`delta`	Token(s) streamed
`tool_call`	Model requested a tool invocation
`tool_result`	Tool execution result returned
`sub_request`	Model dispatching a child HAPPI envelope
`completed`	Response finished
`error`	Failure with classified cause

The Four Axioms:

1. CLI (`stdio`) is the canonical transport — everything else is a mapping.
2. Seven event types are sufficient — no more are needed for v1.0.
3. `sub_request` recurses through the same runtime.
4. The polyglot form IS the specification, not a description of it.

1.2 Auth Portability

The `auth.token` field accepts three forms:

Form	Source	When Used
`env:VAR_NAME`	Shell environment variable	CI/CD, GitHub Actions
`keychain:SERVICE`	macOS Keychain	Developer workstation
Literal string	Inline value	Scripted, short-lived

The auth field is automatically scrubbed from all event stream logs. The same envelope file runs identically on a developer's MacBook and in a GitHub Actions runner — only the auth source changes.

1.3 Transport Layer

HAPPI is transport-agnostic. The reference transport is `stdio`. Additional transports map the same semantics:

Transport	Status	Notes
`stdio`	Canonical (v1.0)	Reference implementation
HTTP/SSE	Supported (v1.0)	Server-sent events
Unix socket	Supported (v1.0)	Low-latency local IPC
WebSocket	Planned (v1.1)	Bidirectional streaming
MCP	Planned (v1.1)	Claude tool integration

No transport adds semantics. A HAPPI client written for stdio works against HTTP/SSE without modification — only the connection establishment changes.

2 The Polyglot Form — The Meta-Move

This is the design property that changes everything. A single `.happi.md` file is simultaneously:

1. **Valid Markdown** — renders as documentation on GitHub, in IDEs, in browsers.
2. **Executable bash** — `bash summarise.happi.md "clause text" | hal -happi-api`.
3. **A HAPPI/1.0 envelope** — the JSON is embedded and extractable.
4. **An OpenAPI 3.1 schema** — machine-readable interface definition.

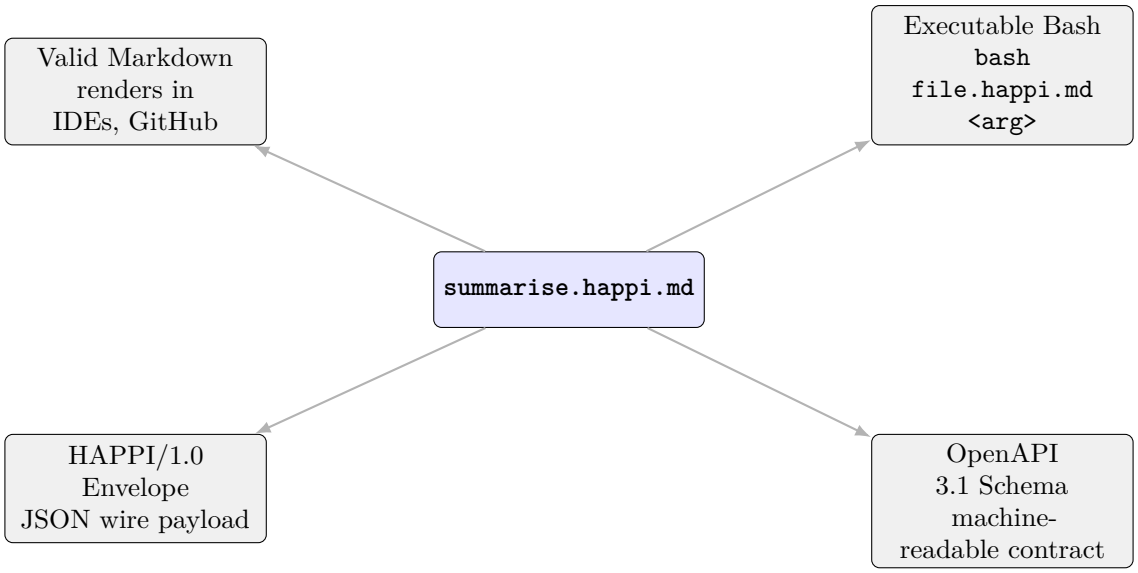

Figure 1: The polyglot form: one source file is simultaneously four artefacts.

Achieved via bash heredoc no-op trick:

```
#!/usr/bin/env bash
# Everything above the heredoc is bash. Everything inside is JSON.
cat <<'HAPPI'
{
  "v": "happi/1.0",
  "id": "summarise-1",
  "cmd": "anthropic.messages.create",
  "args": [{ "model": "claude-opus-4-7",
             "messages": [{ "role": "user",
                            "content": "$1" }] }],
  "auth": { "scheme": "subscription" }
}
HAPPI
# Run: bash summarise.happi.md "Summarise this contract clause: ..."
#        | hal --happi-api
```

2.1 Why This Matters

In every other protocol — HTTP, SMTP, gRPC, TCP — the specification document and the running system are disjoint. RFC 9110 does not itself carry HTTP traffic. HAPPI inverts this. The spec runs itself every morning as boot validation:

```
predict  ->  what should this envelope produce?
observe  ->  run it via hal --happi-api
record   ->  did the output match the prediction?
repeat   ->  next morning
```

Documentation drift is structurally impossible. If the spec diverges from the runtime, the boot cycle fails. The morning report tells you.

2.2 Version-Controlled AI Behaviour

```
git log prompts/summarise.happi.md
# commit abc123: switch to gemini-flash for latency
# commit def456: add brand-voice system prompt
```

```
# commit 789abc: cap max_tokens 2048 -> 4096 for long clauses
```

AI behaviour changes are reviewed, diffed, reverted, blamed, and bisected using the same tools as code. Prompt engineering becomes software engineering.

2.3 Documentation That Cannot Go Stale

A README can lie. An OpenAPI spec can drift from the code. A HAPPI `.md` file cannot — it IS the running artifact. The Markdown is not a description of the system; it is the system reading itself aloud.

3 Multi-Model Orchestration Patterns

HAPPI's `sub_request` event type makes multi-model orchestration a wire-level property, not an application-level concern.

3.1 Sequential Chains

Research, summarise, and critique across different providers:

```
bash research.happi.md   | hal --happi-api \
  | bash summarise.happi.md | hal --happi-api \
  | bash critique.happi.md  | hal --happi-api \
  > verdict.ndjson
```

Each `.happi.md` file is a single-purpose primitive. Composition is `/bin/sh`. No orchestration framework is installed, versioned, or maintained.

Provider strategy per stage:

Stage	Recommended Provider	Reason
Research	`claude-opus-4-7`	High context, deep reasoning
Summarise	`groq/llama-3.3-70b`	Fast, cheap, sufficient
Critique	`google/gemini-2.5-pro`	Different vendor — anti-homogeneity
Verdict	Council deliberation (§3.3)	Stakes justify the cost

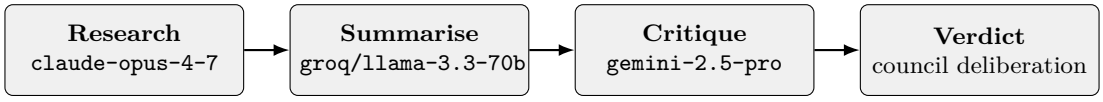

Figure 2: Sequential multi-model pipeline: each stage dispatches to a different provider via the same HAPPI wire protocol.

3.2 Parallel Fan-Out

Same query dispatched to N providers simultaneously; responses merged:

```
parallel -j4 "hal --happi-api < {}" \
  ::: claude.json gemini.json llama.json grok.json \
  | jq -s 'group_by(.request_id)'
```

Applications include: quorum voting, diversity sampling, latency tournament, and provider benchmarking.

3.3 Deliberation Councils (Quorum)

```
{
  "v": "happi/1.0",
  "id": "council-1",
  "cmd": "happi.council",
  "flags": {
    "providers": ["claude-opus-4-7","gemini-2.5-pro",
                  "llama-3.3-70b","grok-3","deepseek"],
    "quorum": 3,
    "rule": "majority",
    "security_override": true
  },
  "args": ["Review this PR for security vulnerabilities."]
}
```

Security issues block on ANY single provider's report. Other issues require quorum. This is a production deliberation pattern currently in use in the authors' internal review pipelines on every pull request.

3.4 Red-Team / Devil's Advocate

One provider proposes; a different provider falsifies:

```
# Stage 1: proposal
echo "Design a token refresh strategy for the auth module." \
  | bash propose.happi.md | hal --happi-api > proposal.ndjson

# Stage 2: adversarial review (DIFFERENT vendor mandatory)
cat proposal.ndjson \
  | bash falsify.happi.md | hal --happi-api > critique.ndjson
```

Same-model-family self-critique suffers from confirmation bias. The falsifier must be a different provider to produce independent signal. This is not a soft recommendation — it is the protocol-enforced invariant in the multi-agent deliberation pattern described in §3.3.

3.5 Cost-Quality Routing (Triage)

Runtime maps complexity class to provider:

Complexity	Provider	Cost (approx.)
Trivial	`groq/llama-3.1-8b`	\$0.00005/query
Moderate	`gemini-2.5-flash`	\$0.001/query
Complex	`claude-opus-4-7`	\$0.015/query
Critical	Council (5 providers)	\$0.05/query

Triage is net-positive when accuracy $> 85\%$ and triage cost $< 5\%$ of mean execution cost. At representative provider catalogue pricing this yields 5–20× cost reduction on mixed workloads.

3.6 Provider Fallback (Sentinel)

```
{
  "flags": {
    "provider": "anthropic/claude-opus-4-7",
    "fallback": ["google/gemini-2.5-pro",
                 "groq/llama-3.3-70b"],
    "fallback_triggers": ["rate_limit", "timeout", "5xx"]
  }
}
```

The client sees a continuous stream with a single `completed`. The provider swap is invisible unless the client reads the `provider` field on each event. The HAL reference runtime's Sentinel (fallback-dispatcher) implementation is ≈200 lines of code. Every application using HAPPI inherits this for free.

4 Polyglot `.md` Workflow Composition

4.1 A Primitive Library

Small, single-purpose `.happi.md` files compose into complex pipelines:

```
primitives/
  research.happi.md       # deep investigation, high-context provider
  summarise.happi.md      # compress to N bullets
  critique.happi.md       # adversarial review
  extract.happi.md        # structured data extraction
  classify.happi.md       # route by category
  score.happi.md          # numerical evaluation
  translate.en-za.happi.md # SA English localisation
  compose.happi.md        # creative synthesis
```

Each primitive is <50 lines. Composition is bash. The "agent framework" is `/bin/sh`.

4.2 Domain-Specific Overlays

```
base/
  research.happi.md            # generic template
domains/
  legal/research.happi.md      # + SA POPI/common law compliance prompt
  medical/research.happi.md    # + HIPAA routing policy, local provider only
  financial/research.happi.md  # + audit trail flags, SOX-compatible output
```

Mechanism: `source base/research.happi.md` in bash; domain overlays mutate the envelope before dispatch. No inheritance framework needed.

4.3 Full Multi-Turn Agent Workflow

```
#!/usr/bin/env bash
# Due-diligence pipeline: 5-stage cross-provider workflow
# Usage: bash due-diligence.sh "target_company_name"

TARGET="$1"

# W1: Research (deep, expensive)
echo "Research $TARGET - financials, litigation, leadership." \
```

```
  | bash primitives/research.happi.md \
  | hal --happi-api \
  | tee /tmp/research.ndjson \
  | bash primitives/extract.happi.md \
  | hal --happi-api > /tmp/facts.ndjson

# W2: Red-team the research (different vendor)
cat /tmp/research.ndjson \
  | bash primitives/critique.happi.md \
  | hal --happi-api > /tmp/critique.ndjson

# W3: Legal clause scan
cat /tmp/facts.ndjson \
  | bash domains/legal/clause-check.happi.md \
  | hal --happi-api > /tmp/legal.ndjson

# W4: Council deliberation
cat /tmp/research.ndjson /tmp/critique.ndjson /tmp/legal.ndjson \
  | bash council/deliberate.happi.md \
  | hal --happi-api > /tmp/verdict.ndjson

# W5: Human-readable summary
cat /tmp/verdict.ndjson \
  | bash primitives/summarise.happi.md \
  | hal --happi-api
```

A five-stage cross-provider multi-model pipeline with no LangChain, no CrewAI, no AutoGen. Bash + HAPPI + HAL.

4.4 Capability Registry (The Marketplace Seed)

```
happi install @codetonight/legal-review-za      # SA jurisdiction
happi install @openai-community/code-critique   # any vendor's community
happi install @internal/brand-voice             # your org's private primitives
happi run legal-review-za --args "contract.pdf"
```

Files are shared across organisations without SDK lock-in. An adopting org runs any published `.happi.md` capability via their own runtime against their own provider keys. Zero publisher SDK dependency.

5 `sub_request` — Native Agentic Recursion

5.1 What It Is

`sub_request` is an event type in the response stream. The model — not the application, not the framework — emits a child HAPPI envelope as part of its response. The runtime dispatches it; child events inline into the parent stream:

```
{"type":"delta",       "request_id":"main",
  "text":"I need to verify this claim..."}
{"type":"sub_request", "request_id":"main",
  "envelope":{"v":"happi/1.0","cmd":"gemini.generate",
              "args":["Cross-check: is it true that..."]}}
{"type":"started",     "request_id":"sub-1"}
{"type":"delta",       "request_id":"sub-1",
  "text":"Cross-model result:"}
{"type":"completed",   "request_id":"sub-1"}
```

```
{"type":"delta",       "request_id":"main",
  "text":"Confirmed. Therefore..."}
{"type":"completed",   "request_id":"main"}
```

5.2 Why This Dissolves Orchestration Frameworks

Every "agent framework" that exists today (LangGraph, AutoGen, CrewAI, Agno, PydanticAI) exists to answer one question: how does one LLM call trigger another LLM call with shared state? `sub_request` answers that at the wire level.

Framework	Recursion	Cross-language	Cross-provider
LangGraph	Python-only	No	Via adapters
AutoGen	Python-only	No	Via adapters
CrewAI	Python-only	No	Via adapters
HAPPI `sub_request`	Protocol-level	Yes	Native

The prediction: "agent framework" as a product category will dissolve the way "RPC framework" dissolved into HTTP + gRPC. Falsified if SDK market share is growing, not shrinking, in 2030.

5.3 Recursive Agent Hierarchies

```
main-request
  +-- sub_request: research (claude-opus-4-7)
        +-- sub_request: fact-check (gemini-2.5-pro)
              +-- sub_request: citation-verify (llama-3.3-70b)
```

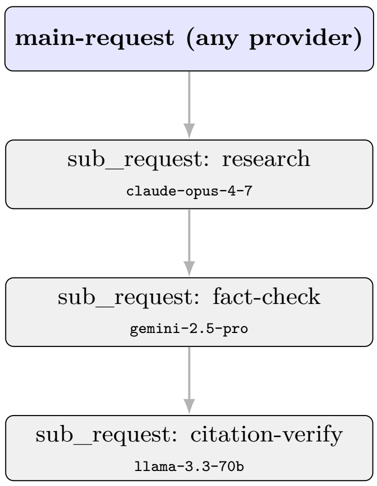

Figure 3: `sub_request` recursion tree: the model dispatches child envelopes; the runtime inlines their events into the parent stream.

Depth cap prevents infinite recursion. The runtime enforces this at the envelope parser. Tree-of-Thought is native: N parallel `sub_request` events at each branch; parent aggregates verdicts. No library needed. The protocol carries it.

5.4 Cross-Language Agent Communication

```
python3 research_agent.py      # Python agent builds research envelope
  | hal --happi-api            # HAL (Rust runtime) dispatches
  | go run summarise_agent.go  # Go agent parses NDJSON
  | hal --happi-api            # HAL again
  | node critique_agent.js     # Node.js agent
  > verdict.ndjson
```

Each agent is in a different language. Each speaks HAPPI. No cross-language RPC. No CORBA. No SOAP. The wire format is the interface.

6 Custom Domain Harnesses

6.1 Legal AI (SA Jurisdiction Pattern)

```
legal-review/
  base.happi.md
  jurisdictions/
    za/clause-check.happi.md    # SA POPI + common law
    eu/gdpr-check.happi.md      # GDPR Article 44 residency
    us/ccpa-check.happi.md      # California privacy law
  handlers/
    conflict-of-laws.happi.md
    citation-verify.happi.md    # OSCOLA / Harvard / Bluebook
    council-review.happi.md     # quorum deliberation for high-stakes
```

Audit trail by design. Envelope flag `audit: true` captures the full event stream to the compliance store. Every `tool_call` is logged. Every provider response is retained. Subpoena-ready without additional instrumentation.

Concrete workflow:

```
for CLAUSE in $(pdfextract contract.pdf --clauses); do
  echo "$CLAUSE" \
    | bash legal-review/jurisdictions/za/clause-check.happi.md \
    | hal --happi-api \
    >> clause-review.ndjson
done
```

Five minutes. Clause-by-clause review. Three providers cross-checking. Full audit trail. Zero third-party data exposure (local Gemma 4 26B for sensitive documents).

6.2 Medical AI (HIPAA-Safe Routing)

The routing policy maps `data_class: phi` to approved on-premises providers only (local Ollama, private-link Claude). Cloud providers are rejected at the envelope layer — no application bug can leak PHI. The guardrail is in the protocol, not in application code that can be bypassed.

6.3 Financial AI (SOX-Compatible Audit)

The NDJSON event stream IS the audit log:

Event	SOX-Relevant Field
`started`	Who queried, when, which provider
`delta`	Full token-level model output
`tool_call`	Every external call (price feed, order entry, risk API)
`tool_result`	Full payload of each call
`completed`	Final verdict, total cost, latency

An auditor replays the event stream to reconstruct every AI-assisted decision. Works identically across GPT-4, Claude, and Gemini — the protocol is neutral, the audit format is consistent.

6.4 Organisation-Specific Prompt Libraries

Changes to `.happi.md` files go through PR review. The prompt library IS the organisation's AI behaviour specification. Version-controlled. Diffable. Reversible.

7 Enterprise and Compliance Implications

7.1 Provider Failover Without Code Changes

Incident response: edit the config file, `SIGHUP` the runtime. Deployment is unnecessary. Application is untouched.

7.2 Data Residency

```
{ "flags": { "data_residency": "eu-west" } }
```

Runtime routes to providers with EU-only inference. Every inference logs provider region in the event stream. GDPR Article 44 compliance becomes a config rule, not an application audit.

7.3 Multi-Tenancy at the Envelope Level

Per-request credential routing. Tenant A's keys never touch Tenant B's requests. At protocol level — not application level.

7.4 Cost Governance

```
{
  "flags": {
    "max_tokens": 2048,
    "max_cost_usd": 0.05,
    "budget_id": "team-legal-2026-q2"
  }
}
```

Runtime rejects if budget exceeded. Monthly reports by aggregating `completed` events — one policy file, all providers.

7.5 Observability Stack (Minimum Viable)

```
HAPPI event stream -> Kafka -> ClickHouse -> Grafana
```

Every runtime produces this for free. Zero instrumentation overhead. The structured event stream IS the telemetry.

8 Developer Experience Transformation

8.1 Any Language Is an AI Application

If you can write and read JSON, you can speak HAPPI. A 1977 awk script is an AI client. Cobol, Fortran, Lua, Tcl, Pascal — all become AI-capable without vendor involvement. This is the syscall property: you do not need Anthropic to support your language; you need `read()`.

8.2 Testability — Mock the Wire, Not the SDK

```python
def test_clause_review():
    with mock_happi_stream([
        {"type": "started",   "request_id": "r1"},
        {"type": "delta",     "request_id": "r1",
          "text": "Ambiguous liability in S4.2."},
        {"type": "completed", "request_id": "r1"}
    ]):
        result = run_clause_review(test_clause)
    assert "S4.2" in result.findings
```

No provider involved. No rate limits. No network. One mock format works for all providers. Compare with SDK mocking: mock each provider SDK separately.

8.3 IDE-Native Development

A `.happi.md` file in an IDE:

- Syntax-highlighted as Markdown.
- Linted by `shellcheck` (bash) + `jq` (envelope) + OpenAPI linter (schema).
- Executable via run button.
- Debuggable by inspecting the NDJSON output stream.

Plugin opportunity: VS Code extension with HAPPI-aware IntelliSense — envelope schema validation, provider catalogue autocomplete, live token cost estimation as you type.

8.4 Onboarding — One Command

```
cat spec.happi.md | hal --happi-api
```

The spec demonstrates itself. New engineer. Day one. Full system behaviour, live, in thirty seconds.

9 "AI is a Syscall" — The Long Arc

9.1 The Structural Claim

Unix `read(fd, buf, len)` does not know whether it reads from SSD, NVMe, RAM, network socket, or `/dev/null`. The application calls one primitive; the kernel dispatches. HAPPI proposes AI inference as this primitive.

The analogy holds structurally, not rhetorically:

Unix syscall	HAPPI equivalent
`read(fd, buf, len)`	`hal -happi-api < envelope.json`
File descriptor	Provider identifier in envelope
Buffer	NDJSON event stream
Kernel VFS layer	HAPPI runtime (HAL, the authors' reference implementation, in development and e
Block device driver	Provider adapter
Physical disk	The actual model (Claude, Gemini, Llama)

9.2 `/dev/ai` — The Concrete Proof-of-Concept

```
echo '{"v":"happi/1.0","cmd":"anthropic.messages.create",
       "args":["Hello"]}' > /dev/ai
cat /dev/ai      # NDJSON events stream out
```

Implementation: ≈500-line FUSE module. Not a 10-year arc — this is a weekend project once HAPPI v1.0 runtime adoption reaches critical mass.

9.3 Shell Scripts with AI as a Pipe Stage

```
git log --oneline -20 \
  | hal chat --provider groq/llama-3.3-70b "Summarise these commits:" \
  | tee weekly-update.md

# Or as a full pipeline:
cat contracts/*.pdf | pdftext | ai-review | jq '.blockers[]'
```

AI reasoning takes its place alongside `grep`, `awk`, `sed`, `sort`, `jq` as a standard Unix primitive. Falsified if cost remains above $0.001/query or latency remains above 500ms for small queries in 2030.

9.4 IoT and Embedded Systems as AI Clients

An ESP32 microcontroller (32KB RAM, no Python runtime) emits HAPPI envelopes over MQTT to a local HAL gateway. AI is now accessible from hardware that cannot run any existing AI SDK.

10 Ecosystem and Marketplace Implications

10.1 Runtime Proliferation

HAL is the authors' reference runtime for HAPPI, currently in development and evaluation. Community runtimes are expected to emerge the way HTTP clients emerged after RFC 2616:

Language	Package	Estimated ETA
Python	hal-py	3–6 months
Node.js / Deno / Bun	hal-js	3–6 months
Go	hal-go	6–12 months
Ruby	hal-rb	6–12 months
Java / Kotlin	hal-jvm	12–18 months
Swift	hal-swift	12–18 months

Each is an implementation of HAPPI/1.0. All interoperate. HAPPI does not need to ask permission from any of these language communities.

10.2 Provider Certification

"HAPPI-compliant" as a standard mark — analogous to "ACID-compliant" or "POSIX-compliant". A provider's HAPPI adapter passes the conformance suite or does not carry the mark.

10.3 HAPPI Routers — The DNS Analogy

HAPPI routers are to AI capabilities what DNS is to IP addresses: capability-resolution infrastructure that the application never sees.

10.4 Cross-Org Workflow Sharing

Organisation A ships `morning-brief.happi.md`. Organisation B runs it via their own runtime against their own provider keys. Zero publisher SDK dependency. The capability travels as a file. The protocol is the interface.

11 What HAPPI Is Not (Precision Matters)

Category	Examples	Distinction from HAPPI
Application framework	LangChain, LlamaIndex, AutoGen	Live inside your application; language-specific
Provider SDK	OpenAI SDK, Anthropic SDK, LiteLLM	Language-specific library; normalise to one format
Proxy server	LiteLLM proxy, OpenRouter, Portkey	Translation layer in network path; not a protocol
Model	Claude, GPT-4, Gemini, Llama	The inference artifact; HAPPI is what speaks to it
Runtime	HAL	The authors' reference runtime for HAPPI, in development and evaluation; not the protocol itself

The precise LiteLLM comparison: LiteLLM is `requests` (the Python HTTP library). HAPPI is HTTP. `requests` is excellent software. It did not make HTTP redundant. They operate at different levels of the stack. LiteLLM could implement HAPPI as its wire format. A HAPPI gateway could accept LiteLLM-proxy traffic via adapter. They are not competitors — they occupy adjacent layers.

12 Honest Limitations and Open Questions

12.1 Where the Protocol Breaks Down

Long-running stateful sessions. HAPPI is request/response-oriented. Applications needing persistent multi-turn conversation with large session state must encode that state in envelopes. Mitigation: `session_id` envelope flag + runtime-level session store. Design tension: this pushes state into the runtime, which complicates pure-dispatch semantics.

Bidirectional real-time streaming. Low-latency voice agents require WebSocket bidirectional streams. HAPPI v1.0 handles unidirectional well; bidirectional is addressed in planned v1.1 WebSocket transport.

High-throughput batch. Processing 10M rows via HAPPI envelope-per-query has overhead. Batch APIs process at 50% cost discount. v1.1 may introduce a batch envelope type.

Provider-specific capabilities. Anthropic prompt caching, OpenAI structured outputs, Gemini context caching — HAPPI's common abstraction can hide these. Mitigation: `flags.provider_specific` passed opaquely to adapter.

12.2 Falsification of the TCP/IP Analogy

The central claim — that HAPPI will do for AI what TCP/IP did for networking — is falsified if:

1. Provider-specific SDK market share is growing, not shrinking, by 2030.

2. Major cloud providers block protocol-level interop and succeed.
3. HAPPI adoption stalls at < 5 production runtimes outside CodeTonight within 18 months.
4. Semantic incompatibility between providers proves irreducible.
5. Regulatory action mandates provider-specific attestation incompatible with protocol-level neutrality.

Current probability assessment: 30–40% the TCP/IP analogy fully holds; 30–40% partial hold (HAPPI becomes a niche standard); 20–30% it fails (SDK lock-in persists). HAPPI/1.0 was first specified on 18 April 2026 — the protocol is days old at time of writing. All falsifiability is prospective.

12.3 Open Protocol Questions for v1.1

Question	Council Vote	Notes
`state` as first-class event type?	5y / 5n / 2abs	Enables cross-runtime state; risks scope creep
`ensemble_synthesis` at protocol level?	3 pro / 7 app	Keep application-level until canonical algorithm exists
Dedicated batch envelope type?	8y / 3n	Strong signal
WebSocket elevated to normative?	9y / 2n	Voice agents demand it
`provider.<id>.<key>` convention?	10y / 1n	Near-consensus
Signing/provenance in envelope?	4y / 7 eco	Security-conscious ecosystem over baked-in complexity

13 Adoption Roadmap

Phase	Timeline	Milestones
Seed	Now – 6 months	HAL reference runtime; happi.md domain live; 3 community runtime seeds (Python, Go, Node)
Ecosystem	6–18 months	20+ provider adapters; capability marketplace seed (100+ .happi.md files); first non-CodeTonight production adoption
Enterprise	18–36 months	Commercial runtime with SLAs; HAPPI certification programme; HIPAA/GDPR/SOX compliance profiles
Standard	3–5 years	Protocol-level market share exceeds SDK market share; HAPPI-native IDEs, linters, CI integrations
Primitive	5–10 years	`/dev/ai` proof-of-concept; AI reasoning as a Unix toolkit member

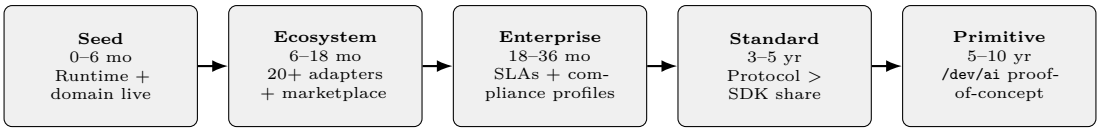

Figure 4: HAPPI adoption roadmap: five phases from reference runtime to Unix-level primitive.

A The Envelope at a Glance

```
{
  "v":    "happi/1.0",
  "id":   "unique-call-id",
  "cmd":  "anthropic.messages.create",
  "args": [ ... ],
  "flags": {
    "provider":          "anthropic/claude-opus-4-7",
    "max_tokens":        4096,
    "data_residency":    "eu-west",
    "data_class":        "phi",
    "budget_id":         "team-legal-q2",
    "max_cost_usd":      0.10,
    "fallback":          ["gemini/gemini-2.5-pro"],
    "fallback_triggers": ["rate_limit", "timeout", "5xx"],
    "audit":             true
  },
  "auth": {
    "scheme": "apikey",
    "token":  "env:ANTHROPIC_API_KEY"
  }
}
```

B Provider Catalogue (HAL v1, April 2026)

Provider	Models	Best For
Anthropic	Claude Opus 4.7, Sonnet 4.6	Complex reasoning, long context
Google	Gemini 2.5 Flash/Pro, 2.0 Flash	Speed, multimodal, cross-check
Groq	Llama 3.3 70B, 3.1 70B, Mixtral	Fast inference, cost
DeepSeek	Chat, Coder, Reasoner	Code, Chinese-language
xAI	Grok 3	Real-time web context
OpenAI	GPT-4o, o1, o1-mini	General purpose
Mistral	Mistral Large/Medium	European data residency
Ollama (local)	Gemma 4 26B, Gemma 4 e4B	Zero cost, zero egress, no rate limit

C MESH Council Deliberation Metadata

A MESH deliberation council is a multi-agent review pattern in which each agent analyses the problem from a distinct specialist perspective, and the outputs are synthesised into a consolidated finding. The use of multiple agents reduces single-model blind spots and surfaces a broader range

of considerations than any individual analysis would produce. In this paper, twelve such agents were convened across ten analytical dimensions.

Council composition
12 specialist agents

Deliberation pattern
MESH — parallel specialist agents synthesising into a consolidated verdict

Dimensions covered
10 (multi-model orchestration, polyglot composition, agent frameworks, domain harnesses, enterprise compliance, developer experience, syscall vision, ecosystem, limitations, falsification)

Falsification discipline
All prospective claims carry explicit falsification conditions

Vote records Preserved in §12.3

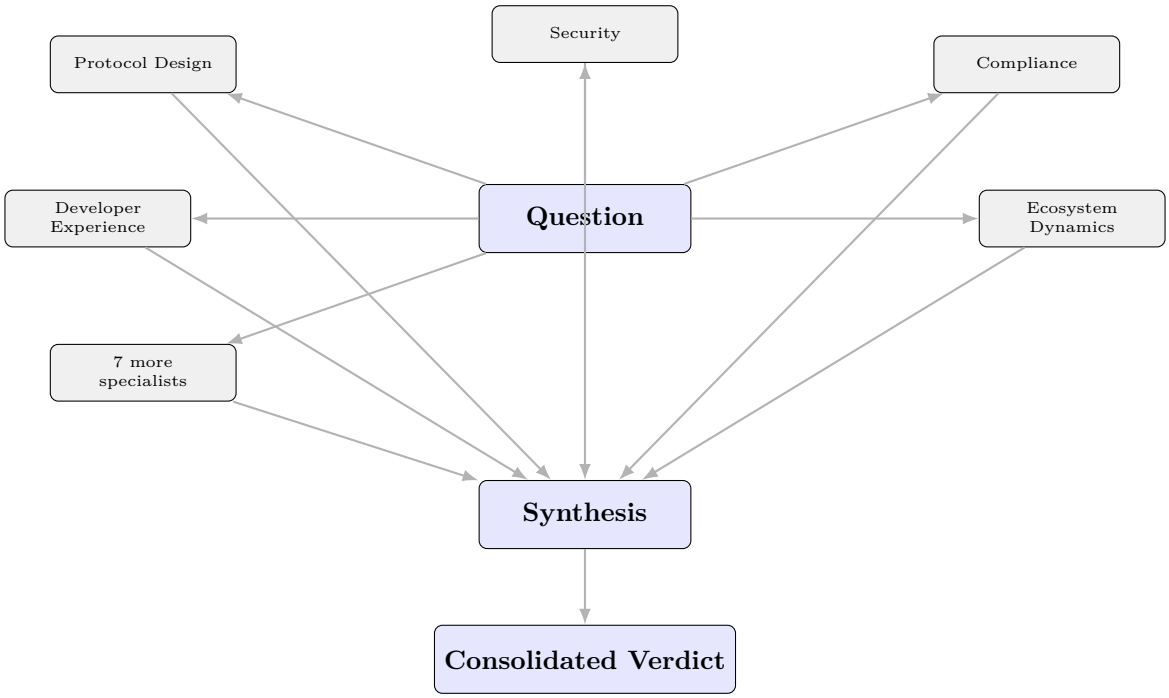

Figure 5: MESH council deliberation: twelve specialist agents each analyse from a distinct perspective; outputs are synthesised into a consolidated verdict.

This paper is a living document. The protocol is stable; the possibilities are not.
Canonical reference: `happi.md`