AIP: A Graph Representation for Learning and Governing Agent Skills

Zach Blumenfeld
Neo4j
MD, USA
zach.blumenfeld@neo4j.com

Jim Webber
Neo4j
London, UK
jim.webber@neo4j.com

ABSTRACT

Agent Skills today consist largely of free-form prose requiring the agent to read, interpret, and re-derive how to act in every session. This imposes two compounding costs: reduced reliability on implementation-heavy tasks, and difficulty in skill creation and improvement. Editing prose is a fragile process that both humans and agents struggle with, particularly for domain-specific procedural knowledge that is underrepresented in model training. The Agent Instruction Protocol (AIP) addresses both by modeling a skill as a directed execution graph with discrete steps as nodes backed by deterministic scripts or natural-language descriptions, connected by explicit typed input/output edges, all governed by a schema-validated YAML specification. A compiler meta-skill translates existing human-written skills into this form.

The benefits are twofold. Firstly, compiling human-written skills to AIP raised Claude Sonnet's mean task reward from 0.60 to 0.71 and pass rate from 53% to 67% across 27 real agent tasks from SkillsBench—a statistically significant gain (Wilcoxon signed-rank $p = 0.011$), winning 12 tasks to 2 with 13 ties. It is often faster since the graph delivers vetted, runnable units to the agent rather than asking it to re-derive code, commands, and tool calls from natural language. Secondly, on creation and improvement, because each skill is schema-validated, functionally testable, and addressable node-by-node, failures can be diagnosed and repaired precisely. Two authored-skill failures were traced to the script level. After adjusting the AIP spec and recompiling, both recovered with zero regressions (one task going from 0/5 to 5/5), turning skill improvement into a measurable tuning loop rather than a prose rewrite. That same graph structure supports corpus-level governance and skill introspection, and provides a natural action space for reinforcement learning over skills.

VLDB Workshop Reference Format:
Zach Blumenfeld and Jim Webber. AIP: A Graph Representation for Learning and Governing Agent Skills. VLDB 2026 Workshop: Agent+Graph.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zach-blumenfeld/aip-skillbench.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

Agent Skills [3], introduced by Anthropic in 2025 and subsequently released as an open standard, have become the dominant representation for packaging reusable agent capabilities. A skill is a directory built around a `SKILL.md` file: YAML frontmatter declaring a `name` and `description`, followed by natural-language instructions, optional scripts, and reference files that the agent loads on demand through *progressive disclosure*. The appeal of the format is that it lets a broad range of domain experts transfer procedural knowledge to AI agents in a lightweight, human-readable form [4]. Moreover, curated skills measurably raise task success across diverse domains [13].

Yet a `SKILL.md` is a (mostly) free-form markdown document, and the skill as a whole is consumed as natural-language content. As a result, the representation inherits several limitations that become acute as agents are deployed on harder, implementation-heavy tasks:

(1) **Skills neither capture nor enforce structure where warranted, leaving speed and reliability on the table.** Some context and judgment resist formalization, but a large share of a skill's procedural knowledge could be expressed as runnable code and explicit graphs of workflow steps, which is increasingly the way reliable agents are built [8, 12, 22]. Skills allow this logic to remain in free prose for the agent to derive at runtime, so on every new session an agent re-plans the code, commands, and tool calls the task needs. On implementation-heavy tasks this is slow and token-intensive, and lets the agent take different, sometimes erroneous, paths that reduce reliability and increase run-to-run variance. Offloading deterministic steps to vetted, runnable code [6, 7] and explicitly structured workflows [22] instead is known to improve reliability. What is missing is a mechanism that reliably compiles this structurable knowledge, including the deterministic steps and the data that flows between them, into vetted, runnable code and an explicit step graph, so the agent need not re-derive it on every run.
(2) **Skill creation and tuning remains a slow, recurring process.** A skill must be tuned like a prompt. A skill consisting of mostly free-form prose is read and re-interpreted by the agent on every run, so it inherits the well-documented brittleness of prompts: small, semantics-preserving changes in wording or formatting can swing task accuracy by tens of points [23] and so it is difficult and error-prone to derive a skill that consistently works correctly.
(3) **Agent-assisted and self-improvement of skills remains a challenge.** A skill is only worth shipping when it encodes

procedural knowledge lacking in model training [4, 13]. Experts therefore author skills, but, because of the challenges presented above, increasingly enlist agents to help *revise* them. The field as a whole is pushing toward agent self-improvement and reinforcement learning over skills [2, 18, 30, 36]. This is hard for two compounding reasons: the material is unfamiliar by construction—the agent is revising domain-specific procedures it does not itself know—and free-form prose gives it no bounded surface to edit against. With nothing to constrain the edit, the agent's additive bias runs unchecked: outputs skew toward length and verbosity [24, 34], and agents over-engineer and accrete rather than refine [15]. Skills balloon and grow convoluted, intent drifts, and a fix in one place silently breaks another, while intrinsic self-revision without external feedback often leaves quality unchanged or degraded [9, 31]. Without a more structured representation, skill improvement has neither a strong natural feedback loop nor a bounded action space.

(4) **Cause, effect, and governance are hard to establish.** Free-form prose has no clear addressable units. When a skill produces a wrong result there is no node, step, or typed value to which the failure can be precisely attributed, and a corpus of prose skills cannot be systematically audited for missing validation or approval steps. This is contrary to the traceability and accountability that the governance of agentic systems at scale demands [20].

The Agent Instruction Protocol (AIP)[1] addresses these issues by modeling a skill as a directed execution graph. Discrete steps become nodes, each backed by a deterministic script or, where human-like judgment is required, a natural-language description. Nodes are connected by typed input/output edges, and the whole structure is governed by a schema-validated YAML specification. A compiler meta-skill allows agents to translate existing human-written skills into this form at authoring time, surfacing ambiguities, type errors, and field inconsistencies before they can cause runtime failures. Crucially, AIP does not displace human authorship. Experts still write the skills, and the compiler translates their human-written source into a typed, script-backed surface that improves how reliably agents *execute* skills today. We also argue this is a better substrate for agent-assisted improvement and reinforcement learning in the future.

We evaluate AIP against human-curated skills on SkillsBench [13], a benchmark of 94 agent tasks across 8 domains, using a stratified 27-task sample with Claude Sonnet as the solver. Compiling skills to AIP produced a statistically significant improvement in task reward (Wilcoxon signed-rank $p = 0.011$), and we found that failures in AIP skills can be diagnosed and corrected at the node level, given coded scripts connected by a clear, typed execution-graph I/O structure, without causing regressions elsewhere. This suggests that the benefits of AIP may compound for agent self-improvement and reinforcement learning (RL), where the typed execution graph provides a bounded, validity-gated action space for learning over skills [25].

Contributions. Our work makes the following contributions:

- **A multi-mode benchmark harness.** An extension to SkillsBench for comparing skill *formats* and skill *authoring methods* under a real solver, enabling controlled head-to-head evaluation.
- **Empirical evidence of improved task reward.** Compiling human-written skills to AIP significantly improves task reward across 27 tasks (mean reward 0.599 → 0.705, +0.106; Wilcoxon $p = 0.011$), with the mechanism localized to executability and procedural consistency.
- **A proof-of-concept for node-level diagnosis and repair.** During early experimentation, two compiler-authored failure modes were diagnosed at the node and script level by an agent (Claude Code). In this case they were corrected by editing the AIP specification, and recovered with zero regressions, but the tooling demonstrates strong potential for a diagnosis–edit–recompile–evaluate loop with a measurable reward signal—a working substrate for agent-assisted improvement and, prospectively, autonomous self-improvement and reinforcement learning over skills.
- **A path to corpus-level governance and inspection.** Because each skill is a typed, schema-validated graph, a library of AIP skills is queryable and inspectable: missing validation or approval steps can be audited, shared sub-procedures discovered, and skills composed from reusable node templates. Projected into a graph database, the same structure supports access control and visual introspection—a skill's steps and their typed input/output rendered as a graph (Figure 2)—moving governance from manual documentation review to a structured query over a typed graph. As agents grow more autonomous and begin running their own self-improvement loops, this inspectable surface is what keeps human oversight and understanding tractable.

2 BACKGROUND

Modern LLM agents carry out tasks by interleaving natural-language reasoning with calls to external tools—running code, querying systems, editing files—and observing the results [21, 32]. Because many tasks recur, these agents increasingly draw on reusable *skills*: packaged procedural knowledge—instructions, and sometimes code—that tells them how to carry out a class of tasks [27]. Anthropic's Agent Skills standardize this as a `SKILL.md` file in a directory of optional resources the agent loads on demand (Figure 1) [1, 3].

A skill's procedure lives largely in the free-form Markdown body of its `SKILL.md` (Figure 1), which the agent reads and interprets at runtime. Though the skill format permits bundled scripts, in practice much of the procedure stays in prose.

Graphs and structured agent workflows. Agent development frameworks already represent agent workflows as graphs whose nodes are discrete steps and whose edges carry the data passed between them. This includes frameworks such as LangGraph and Google's Agent Development Kit (ADK) [8, 12]. Earlier than that, DSPy, a Python framework for building AI systems, used declarative

[1]Today AIP is realized as a *specification*, a schema-validated execution-graph format that an agent reads into context and follows. We retain the term *protocol* for the typed contract this format defines, and for a runtime that would enforce graph traversal and local and remote skill calls. We set that out as future work (Section 6.4).

Skill directory

```
pdf-processing/
  SKILL.md       # required: YAML frontmatter + instructions
  scripts/       # optional: executable code (e.g. extract.py)
  references/    # optional: docs the agent loads on demand
  assets/        # optional: templates, resources
```

SKILL.md

```
---
name: pdf-processing
description: Extract text and tables from PDFs, fill forms,
  and merge files. Use when working with PDF documents.
---
# PDF Processing
## Extracting text
Run `scripts/extract.py <file>`; it writes one block per page...
## Filling forms
See references/forms.md for the field-mapping convention...
```

Figure 1: An Agent Skill is a directory built around a `SKILL.md` file—YAML frontmatter (required `name` and `description`) followed by free-form Markdown instructions—alongside optional `scripts/`, `references/`, and `assets/` that the agent loads only as a task requires (*progressive disclosure*) [1].

pipelines that can be compiled rather than hand-written to automatically optimize how language models are prompted (or fine-tuned), replacing brittle, manually tuned prompt templates [11]. Anthropic, the AI safety and research company behind the Claude family of models and the agent skill spec, also recommends predefined, structured workflows where predictability and reliability matter [22]. Dedicated benchmarks likewise find that supplying agents with explicit workflow structure improves their planning [29]. Evidence that this matters comes from the same benchmark we use, SkillsBench, where the skills that help most pair focused guidance with executable code and reference files, while sprawling, comprehensive prose can actually hurt performance [13]. It is structure, not just content, that shapes how well a skill works. AIP brings this graph view to the skill itself, where a skill becomes an execution graph of typed steps, each backed by a script or a natural-language description.

Measuring skills. We build on SkillsBench, a containerized benchmark that isolates the effect of a skill. Each task pairs a natural-language instruction with a sandboxed environment and a programmatic verifier, run through the BenchFlow SDK [5], and is scored under three conditions: no skill, a human-curated skill, and a skill the agent generates for itself. Measuring task success across these conditions lets a skill's contribution be quantified directly, which our experiments extend with AIP.

3 THE AGENT INSTRUCTION PROTOCOL (AIP)

AIP is an extension of the Agent Skills specification [1], one that defines a skill as a directed execution graph—today a specification, with the enforcing protocol left to future work (Section 6.4). Nodes represent discrete operational steps where each step is either backed by a deterministic script for computational work, or described in natural language for steps requiring judgment or interaction. Step nodes are connected by typed input/output edges that make data flow explicit and checkable, and the entire structure is governed by a schema-validated YAML specification. Figure 2 shows a real compiled skill rendered as a graph. The major node and edge types are as follows:

```
# Nodes
Skill   # the procedure: purpose, trigger_when, scope_and_approval
Step    # a unit of work: name, description; an optional backing
        #   script; typed inputs/outputs; depends_on/parallel/one_of

# Edges between steps
inputs / outputs   # a step's typed output (name, type) feeds another
                   #   step's input -- the typed data-flow edges
depends_on         # explicit ordering for DAG-shaped procedures

# A step also binds package files
script             # a deterministic body under scripts/
references         # prose-cited docs under references/
```

Steps may also carry `parallel` and `one_of` control modifiers. The remaining top-level metadata—trigger and do-not-use conditions, anti-patterns, scenarios, modes, and integrations—attach as typed satellite nodes, keeping the entire specification queryable (Section 6.2).

Compilation. AIP includes a compiler meta-skill that transforms human-written source material such as existing skills, prose instructions, documentation, code, or informal descriptions into the graph representation. The compilation step acts as a quality gate where schema validation catches type errors, missing fields, and structural inconsistencies at authoring time rather than at runtime. Ambiguities in the source material must be resolved to produce a valid graph, which forces clarity that prose allows to remain implicit. The meta-skill also provides instruction on creating scripts from natural language where appropriate, which we have found can be a load-bearing step for improving reward. Some of the largest gains in our evaluation (Section 4) are delivered by prose-heavy skills where the solver would otherwise re-derive code at run time, such as `mars-clouds-clustering` and `dapt-intrusion-detection`. Figure 3 shows this transformation on a real skill, contrasting the human-curated and compiled on-disk packages.

Execution. At runtime, the agent loads the `SKILL.md` graph into its context window where it can reason through the execution graph steps. With more scripts and explicit input/output between steps, the agent can re-focus language-model reasoning on the prose nodes that genuinely require judgment and avoid re-deriving well-understood procedures on every run. The same principle of offloading deterministic computation to code is well-known to improve reliability [6, 7].

Addressability. Because each node is individually named, typed, and validatable, the graph is addressable at the component level. A failure in execution can be attributed to a specific node, the corresponding script can be inspected or corrected, and the repair can be validated in isolation. This property underlies both the skill-improvement loop evaluated in Section 4 and the governance, learning, and protocol roadmap outlined in Section 6.

4 EVALUATION

4.1 Experimental Setup

Benchmark. We build on SkillsBench, a containerized agent benchmark of 94 tasks across 8 domains, run through the BenchFlow SDK [5]. Each task bundles a natural-language instruction, a sandboxed environment, and a programmatic verifier that scores the

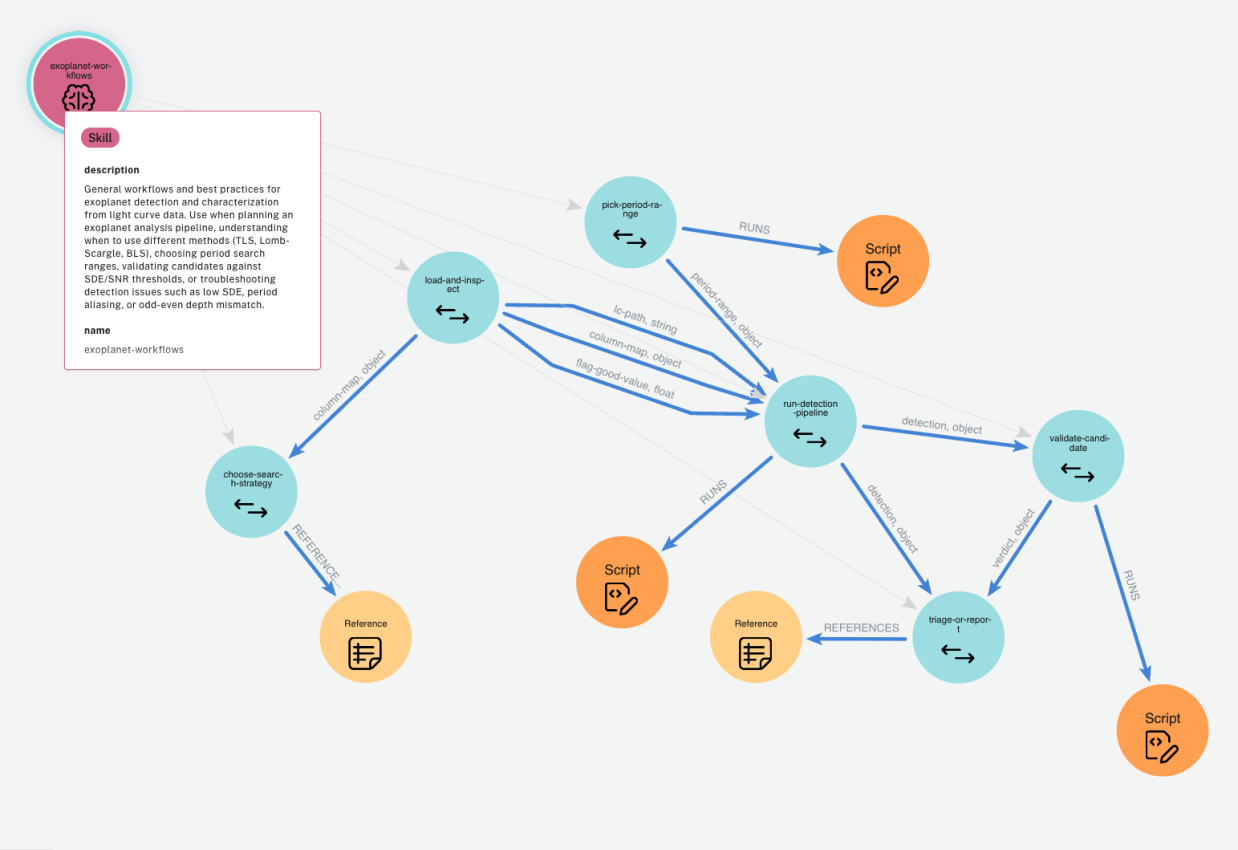

Figure 2: A compiled AIP skill (`exoplanet-workflows`) as a directed execution graph, projected into Neo4j. The pink Skill node carries the name and description; teal nodes are procedure *steps*; arrows between steps are *typed input/output edges* (e.g. `lc-path, string`; `detection, object`) that make data flow explicit and checkable. A Runs edge binds a step to a deterministic *script* (orange); a References edge attaches a prose *reference* (yellow) for steps needing judgment. The same typed structure is what makes a skill node-addressable, and queryable (Section 6.2).

agent's output. SkillsBench is designed to measure how an agent's task success changes when it is given reusable skills, and ships three native conditions per task: 1) *noskill*, 2) one or more *human-curated* skills authored offline by a domain expert, and 3) *self-generated* one or more skills the agent writes for itself at trial time.

We extend SkillsBench with a multi-mode harness that adds two AIP conditions: 1) *aip-from-instruction* where one or more AIP skills are authored by a Claude Code Agent from the task instruction alone, and 2) *aip-from-curated* where one or more AIP skills are authored by a Claude Code Agent using the human-curated skill(s) as input. Both modes use the AIP meta-skill to compile the skills and both use Opus 4.7 as the language model.[2] Following AIP's author-once, consume-many design, each AIP skill is authored a single time, committed, and mounted unchanged across all trials. The harness thus enables controlled head-to-head comparison of AIP skill *formats* and *authoring methods* under a single solver.

Solver. The agent harness is `claude-agent-acp`, driving the `claude-sonnet-4-6` model, with all executions isolated in a Docker sandbox and five independent trials per cell (one task under one condition).

Conditions. The primary comparison is between the *human-curated* and *aip-from-curated* modes. The harness modes (*noskill* and *selfgen*) are out of scope of the main claim. SkillsBench already found *human-curated* outperforms both these modes on average. *aip-from-instruction* was attempted in early trials, but results were excluded from this report. In the v0.3a2 medium set, *aip-from-instruction* performed substantially lower than the human baseline (0/15 passing, vs. 11/15 for human-curated), with failures traced to incorrect methods in scripts committed into the authored skill—a distorting map projection, a parser that rejected valid inputs, a controller that produced no output. This reiterates SkillsBench's finding that self-authored skills provide no benefit on average—models cannot reliably author the procedural knowledge they benefit from consuming [13]—and that effective skills rest on expert curation [4]. It also leans toward the AIP format, rather than the authoring model

[2]The AIP protocol—its specification, schemas, and compiler meta-skill—is open source on GitHub at https://github.com/zach-blumenfeld/aip (the results in this paper use tag `v0.3a3`). The AIP-SkillBench harness, run data, and analysis are available on GitHub at https://github.com/zach-blumenfeld/aip-skillbench.

Before: human-authored prose (`SKILL.md`)

```
# Exoplanet Detection Workflows
## Pipeline Design Principles
### Key Stages
1. Data Loading: format, columns, time system
2. Quality Control: filter via quality flags
3. Preprocessing: remove noise, preserve signal
4. Period Search: choose algorithm for signal
5. Validation: verify candidate is real
6. Refinement: improve period precision

### Critical Decisions
Which period search algorithm?
- TLS: best for transit-shaped dips (box-like)
- Lomb-Scargle: any periodic signal, fast
- BLS: alternative to TLS, in Astropy
```

After: compiled AIP procedure (`SKILL.md`, YAML)

```
purpose: >
  Load, quality-filter, preprocess, search for a transit
  signal, validate vs SDE/SNR thresholds, refine period.
steps:
  - name: load-and-inspect
    outputs:
      - {name: lc-path,         type: string}
      - {name: column-map,      type: object}
      - {name: flag-good-value, type: float}
  - name: choose-search-strategy   # judgment node
    description: >
      Default to TLS; load references/method-selection.md
      to justify TLS vs Lomb-Scargle vs BLS.
    one_of:  [tls, lomb-scargle, bls]
    outputs: [{name: method, type: string}]
  - name: run-detection-pipeline   # script node
    script:  scripts/detect_period.py
    inputs:  [lc-path, column-map, flag-good-value]
    outputs: [{name: detection, type: object}]
  - name: validate-candidate       # script node
    script:  scripts/validate_candidate.py
    inputs:  [detection]
    outputs: [{name: verdict, type: object}]
```

On-disk skill package (before → after)

```
# Before: human-curated (prose-only)
exoplanet-workflows/
  SKILL.md     # prose guidance only
```

```
# After: compiled AIP (exoplanet-workflows)
exoplanet-workflows/
  SKILL.md            # AIP procedure (YAML above)
  scripts/            # deterministic step bodies
    detect_period.py
    period_range_guide.py
    validate_candidate.py
  references/         # prose for judgment steps
    method-selection.md
    troubleshooting.md
  source/             # provenance kept by compiler
    original-SKILL.md
    procedure.schema.json
```

Figure 3: Compiling a prose skill to AIP, for `exoplanet-workflows`. The free-form procedure (top left) becomes a schema-validated YAML graph of typed steps (top right): each step declares typed `inputs`/`outputs`, binds to a deterministic script via `script`, or cites a prose `reference` from its description for judgment. On disk (bottom), the prose-only human skill—`exoplanet-workflows`, one of five modules the task bundles, all shipping no code—expands into a package whose steps are backed by generated `scripts/` and `references/`, with the original prose and JSON schema preserved under `source/`; Figure 2 is this same skill projected as a graph in Neo4j.

(Claude Opus 4.7) being the source of the uplift in *aip-from-curated*, though this does not fully isolate the two (see Limitations 4.5).

Task sample and strata. We evaluate on 27 tasks, listed in Table 1. We characterize each along three axes:

- **Difficulty** (*easy*, *medium*, or *hard*): the task author's rating, taken directly from SkillsBench task metadata.
- **Implementation class** (*light* or *heavy*): a task is *heavy* when its declared type includes implementation, simulation, optimization, or control, i.e. it requires substantial code to be written or run; it is *light* otherwise (analysis, calculation, extraction, search, or detection).
- **Structure class** (*prose-only*, *mixed*, or *script-heavy*): a property of the *human-curated* skill: how much runnable code it already ships. A *prose-only* skill has no scripts (all natural language); a *mixed* skill has scripts but more prose than code (measured in LoC); a *script-heavy* skill has at least as much script as prose. This axis indexes how much room an AIP conversion has to add executability: most for prose-only skills, least for script-heavy ones.

The 24-task core is stratified across structure class × implementation class so the evaluation spans the gradient of expected AIP benefit rather than a single operating point, with difficulty and domain as secondary spread; all 24 use AIP v0.3a3. To this we add a three-task medium reference set (*eval-3med*, marked † in Table 1). We report the full 27-task sample as the headline result, with the caveat that those three were compiled against an earlier protocol version (v0.3a2) and randomly selected (from the pool of medium difficulty tasks) rather than stratified. They were an earlier trial of experimentation. Results restricted to the 24-task v0.3a3 set are consistent in direction and significance (Section 4.5).

Metrics. The primary metric is mean task reward, which is robust to the ceiling and floor effects introduced by all-or-nothing verifiers. Secondary metrics are pass rate, wall-clock execution time, and tool-call count.

4.2 Experimental Results

Table 2 reports the aggregate comparison between human-curated skills and their AIP-compiled counterparts. Compiling to AIP raises

Table 1: The 27 evaluation tasks, grouped by the structure class of their human-curated skill (the axis indexing AIP's room to add executability). *Diff.* is the SkillsBench difficulty rating; *Impl.* is the implementation class (*heavy* = the task type includes implementation, simulation, optimization, or control); *Skills* is the number of curated skill modules the task bundles. The 24-task core is stratified across structure × implementation class; † marks the three supplementary *eval-3med* tasks (v0.3a2 spec, hand-picked).

Task	Domain	Diff.	Impl.	Skills	Description
Prose-only — human skill ships no scripts (12 tasks; most AIP room)					
`energy-unit-commitment`	Industrial/Physical Systems	hard	heavy	3	Schedule generator unit commit for day-ahead demand.
`mars-clouds-clustering`	Natural Science	hard	heavy	3	Optimize unsupervised clustering of Mars cloud observations.
`adaptive-cruise-control`	Industrial/Physical Systems	med.	heavy	5	Implement and simulate an adaptive-cruise-control law.
`bike-rebalance`	Mathematics/Formal Reasoning	med.	heavy	4	Plan the optimal overnight redistribution of shared bikes.
`drone-planning-control`†	Industrial/Physical Systems	med.	heavy	6	Generate drone trajectories and feedback control in simulation.
`parallel-tfidf-search`	Software Engineering	med.	heavy	3	Implement a parallelized TF-IDF document search.
`fix-build-google-auto`	Software Engineering	easy	light	3	Repair build errors in a Java codebase so it compiles.
`offer-letter-generator`	Office/White-Collar	easy	light	1	Fill a `.docx` offer-letter template with a conditional block.
`enterprise-information-search`	Office/White-Collar	hard	light	1	Answer a retrieval query over enterprise documents.
`spring-boot-jakarta-migration`	Software Engineering	hard	light	5	Migrate a Spring Boot codebase to Jakarta EE namespaces.
`earthquake-plate-calculation`†	Natural Science	med.	light	1	Find the in-plate quake farthest from the Pacific boundary.
`exoplanet-detection-period`	Natural Science	med.	light	5	Detect an exoplanet and compute its orbital period.
Mixed — some scripts, but prose dominates (9 tasks)					
`energy-market-pricing`	Industrial/Physical Systems	hard	heavy	4	Clear an energy market and compute locational prices.
`grid-dispatch-operator`	Industrial/Physical Systems	med.	heavy	3	Compute an economic unit dispatch for a grid.
`jax-computing-basics`	Software Engineering	med.	heavy	1	Implement numerical routines in JAX.
`suricata-custom-exfil`	Cybersecurity	med.	heavy	3	Author a Suricata rule for a custom exfiltration pattern.
`court-form-filling`	Office/White-Collar	easy	light	1	Extract case data and fill a court form.
`powerlifting-coef-calc`	Office/White-Collar	easy	light	3	Compute powerlifting scoring coefficients.
`dapt-intrusion-detection`	Cybersecurity	hard	light	2	Detect advanced-persistent-threat intrusion in PCAP traffic.
`crystallographic-wyckoff-pos.`†	Natural Science	med.	light	2	Wyckoff position analysis from X-ray CIF files.
`protein-expression-analysis`	Natural Science	med.	light	1	Analyze cancer cell-line protein-expression data.
Script-heavy — skill is already executable and terse (6 tasks; least AIP room)					
`dialogue-parser`	Software Engineering	easy	heavy	1	Parse dialogue text into a structured format.
`civ6-adjacency-optimizer`	Mathematics/Formal Reasoning	hard	heavy	4	Optimize district adjacency placement on a Civ VI map.
`data-to-d3`	Software Engineering	med.	heavy	1	Build a D3.js (v6) visualization of stock data.
`3d-scan-calc`	Industrial/Physical Systems	hard	light	1	Calculate the mass of a 3D-printed part from its geometry.
`sec-financial-report`	Finance/Economics	hard	light	2	Search SEC filings and analyze a financial report.
`travel-planning`	Mathematics/Formal Reasoning	med.	light	6	Plan an itinerary under scheduling constraints.

mean task reward from 0.599 to 0.705 (+0.106), a statistically significant gain under a Wilcoxon signed-rank test ($p = 0.011$), winning 12 tasks against 2 losses with 13 ties. Pass rate rises in parallel, from 53.3% to 67.4%. The 24-task v0.3a3 subset, which excludes the three hand-picked supplementary tasks, is consistent in direction and significance (+0.101, $p = 0.022$), confirming the headline does not hinge on the supplementary set. Figure 4 shows the trial-level outcomes behind these aggregates: the gains are concentrated in a subset of differentiating tasks, where compiling to AIP converts failing or timed-out trials into passes, while the many tied tasks reflect mutual ceilings (both formats pass all five trials) or mutual floors (both fail).

4.3 Execution time

AIP skills also run faster on average. Mean wall-clock time falls from 585 s to 510 s, and AIP is the faster format on 16 of 27 tasks—but this aggregate reduction is *not* statistically significant (Wilcoxon $p \approx 0.28$ two-sided, and $p = 0.14$ under the directional hypothesis that AIP is faster). We therefore treat wall-clock and tool-call counts as descriptive, not as significance claims. The speedups are concentrated on a minority of tasks where prose forced the solver to re-derive code at run time (e.g. `dapt-intrusion-detection` 2/5 → 5/5 at ∼2.2× faster, `jax-computing-basics` ∼2.3× faster).

Figure 5 breaks the per-task wall-clock change down by stratum. The clearest pattern is along `structure_class`: prose-only skills, where AIP has room to add executable structure, tend to speed up, whereas already-terse script-heavy skills do not—mirroring the reward analysis, in which no single structural axis reaches significance.

Four tasks sit far outside the bulk of the distribution and are individually instructive; together they show that a wall-clock change is meaningful only when read alongside the reward outcome.

`mars-clouds-clustering` (1305 secs faster). The human skill is prose-only (309 lines of reference text, no scripts) for a task that requires an 847-combination grid search over a clustering pipeline, scored by an all-or-nothing verifier. The human-curated solver re-derives the full pipeline on every trial and passes only 2 of 5: two completed runs produce a wrong result on an exact-specification detail, and a third computes the correct answer but is killed after idling at the harness time limit. The AIP conversion ships the vetted procedure and passes 5 of 5 while running roughly 30% faster at equal load. The headline 1305 secs slightly overstates the speedup, as two AIP trials ran under lighter concurrency.

`drone-planning-control` (546 secs faster). Also prose-only (463 lines, no scripts). The human solver re-derives the trajectory-and-control stack from prose and passes 2 of 5, with one run stalling

Figure 4: Per-trial outcomes for all 27 tasks under the two skill formats (five trials per task; within each cell the markers are sorted pass, fail, timeout), with the per-task mean reward and mean wall-clock (seconds) over those five trials shown to the right of each block. Each row is a task; the left block is the human-curated skill and the right block its AIP-compiled counterpart. A pass denotes a passing verifier (reward = 1); all timeouts are trials terminated at the harness wall-clock cap or idle limit and score as reward 0, so the means are taken over all five trials. The higher mean reward of the two formats is shown in bold. Tasks are grouped by the AIP spec version their packs were compiled against and sorted alphabetically within each group: the 24-task stratified cohort (v0.3a3) and the three-task supplementary reference set (v0.3a2). Differentiating tasks (e.g. `dapt-intrusion-detection`, `mars-clouds-clustering`, `exoplanet-detection-period`) show failing or timed-out human-curated trials converted to passes after compilation; tied tasks are mutual ceilings or floors that no packaging could move.

Table 2: Human-curated vs. AIP-compiled skills on SkillsBench (Claude Sonnet solver, 5 trials/task). Reward is the primary metric; Δ is AIP − human. The Wilcoxon signed-rank test is computed on per-task mean reward (`scipy` [26] default, tied pairs dropped). Wall-clock and tool-call deltas are descriptive: their aggregate differences are not statistically significant.

Metric	Human	AIP	Δ
27-task headline sample			
Mean task reward	0.599	0.705	+0.106
Pass rate	53.3%	67.4%	+14.1 pp
Mean wall-clock (s)	585	510	−75[†]
Mean tool calls	27.7	25.6	−2.1[†]
Win / tie / loss		12 / 13 / 2	
Wilcoxon p (reward)		**0.011**	
24-task v0.3a3 subset (robustness)			
Mean task reward	0.567	0.668	+0.101
Pass rate	50.8%	63.3%	+12.5 pp
Win / tie / loss		10 / 12 / 2	
Wilcoxon p (reward)		**0.022**	

[†]Not statistically significant (per-task Wilcoxon $p \approx 0.28$); the mean reduction is driven by a few tasks. Pass rate is the fraction of trials with a passing verifier; mean reward is preferred because several verifiers are all-or-nothing, producing the high tie count.

at the idle limit and two earning only partial credit; the AIP version passes 5 of 5 with markedly lower and tighter wall-clock.

`fix-build-google-auto` (finished 430 secs faster, by doing less work). A build-repair task that both formats essentially fail (mean reward 0.20 each): both arms thrash through 90–140 tool calls and exhaust the execution budget on multiple trials. AIP's lower mean wall-clock is an artifact of its unsuccessful trials terminating earlier, rather than a genuine efficiency gain—here the skill is not the bottleneck.

`energy-market-pricing` (511 secs slower but more accurate). The lone task where AIP is markedly slower. The conversion added a heavier computational path that lands more correct results (4 of 5 vs. 3 of 5) but at roughly twice the tool calls and wall-clock. AIP's executability lever buys reliability at a compute cost that is not uniform across tasks.

4.4 Skill improvement

Our results demonstrate that AIP skills are more *executable*. They are also more *improvable*, and for the same reason: because an AIP skill is a graph of named, typed, schema-validated nodes—each backed by a script that can be run and tested in isolation—a failure can be localized to a specific node and repaired easily, rather than by rewriting prose and hoping. We observed this loop directly while iterating the protocol from v0.3a2 to v0.3a3. Two skills that the compiler had authored with latent defects were diagnosed at the script level by an agent (Claude Code), corrected by a change to the AIP meta-skill, recompiled, and re-evaluated.

`offer-letter-generator`: 0/5 → 5/5. The compiled skill contained a frozen conditional-key bug, a template lookup keyed on `RELOCATION` rather than `RELOCATION_PACKAGE`, so every trial failed in the same way. The defect was localized to a single node and fixed

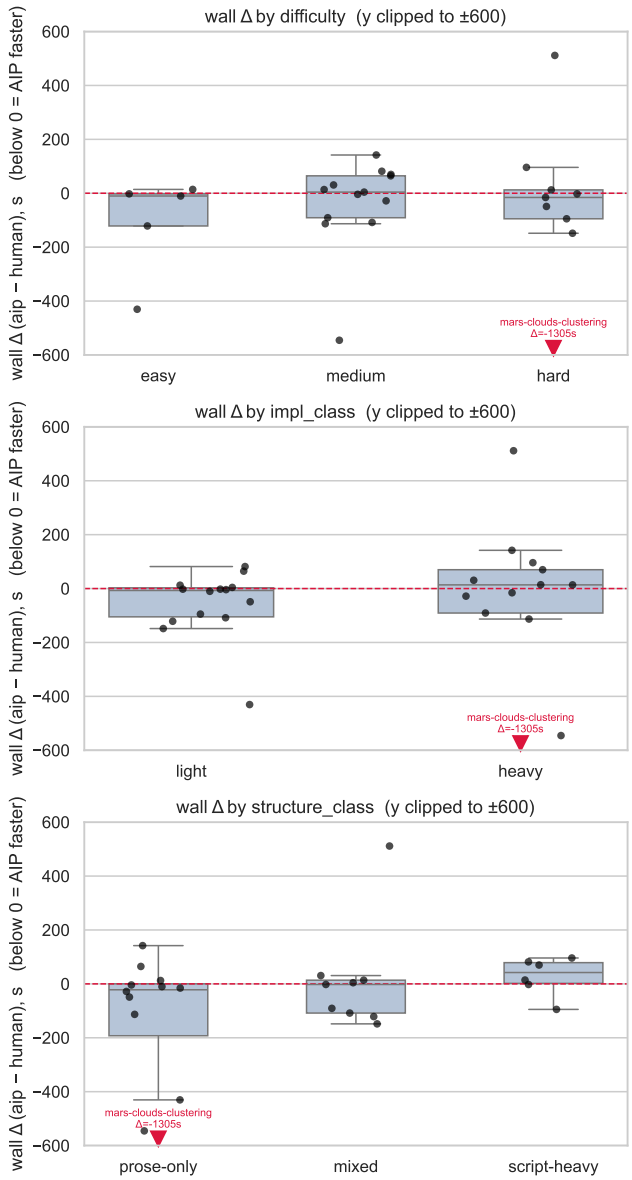

Figure 5: Per-task wall-clock change from compiling to AIP (*aip* minus *human* mean over five trials, in seconds) within each stratum; below the dashed line means AIP is faster. Boxes show the interquartile range and median; dots are individual tasks. The y-axis is clipped to ±600 s, so `mars-clouds-clustering` (−1305 s) is off-scale and flagged with a marker in the `hard`, `heavy`, and `prose-only` panels.

by a specification change (a functional-test correctness check, plus a key-suffix fallback and a default-keep rule) in this case. After recompilation the skill passed all five trials.

`bike-rebalance`: 0.40 → 0.60, timeouts 3 → 1. The compiler authored an over-engineered routing script (roughly 1,146 lines)

heavy enough to exhaust the agent's time budget on three of five trials. A specification change favoring lean scripts produced a smaller routine that fit the budget, cutting timeouts and lifting reward.

Both repairs were verified to cause *zero regressions* on the remaining tasks. The node-level edit fixed the target skill without disturbing the rest of the corpus. We executed these fixes by editing the AIP meta-skill and recompiling, but the loop is not bound to the meta-skill. The diagnosis was already agent-driven, and because each skill is a bounded, typed, addressable artifact—not just prose—the edit itself is a constrained, checkable action an agent can take directly on the skill, rather than the open-ended language editing where agents perform poorly. A team adopting AIP can therefore hand more skill maintenance to its own agents. Diagnosis, specification edit, recompilation, and re-evaluation thus form a closed feedback step with a measurable reward signal—which we argue in Section 6.3 is the natural substrate for reinforcement learning over skills.

4.5 Limitations

The evaluation, while yielding a statistically significant result, carries several important caveats that qualify the strength of our claims.

Format–author confound. The evaluation measures a compile-then-run pipeline: an agent compiles a human-written skill to AIP and then executes it. A performance gain could therefore reflect the improved graph representation, or it could reflect improvements made by agent scripting capability. The two mechanisms are not yet measured fully separately.

That said, the AIP spec—with typed inputs and outputs—shapes script design and appears to contribute to evaluation and improvement. The two node-level repairs of Section 4.4 appear to be guided by the typed specification: because the spec isolated and sequenced steps as connected nodes, the agent could localize the brittleness to a specific script and address it through a functional-testing rule in the meta-skill. This suggests the typed representation, not merely scripting, carries part of the effect. A clean separation nonetheless requires the ablation described in Section 6.1.

Statistical power. Trials are limited to $n = 5$ per cell, and the Wilcoxon signed-rank test operates on per-task means over 12–14 non-tied pairs. The result is significant but preliminary and any broader claims should await a wider trial budget and a larger task set.

Verifier characteristics. Several tasks use all-or-nothing verifiers, which inflates per-task variance and produces the high number of ties (12–13). A finer-grained reward signal would provide more discriminative power and reduce ceiling and floor effects.

Subset caveats. Three of the 27 headline tasks (the *eval-3med* reference set) were compiled against an earlier protocol version (v0.3a2 rather than v0.3a3) and were randomly sampled from medium difficulty tasks rather than drawn from the stratified sampling procedure. We therefore report both the full 27-task figure and the 24-task v0.3a3 subset, and the two are consistent in direction and significance: +0.106 mean reward at $p = 0.011$ over 27 tasks, versus +0.101 at $p = 0.022$ over the 24-task stratified set. Accordingly, the headline result does not depend on the supplementary tasks.

Budget-capped tasks. Two tasks (energy unit commitment and Civilization 6) hit the execution budget without producing informative results. They are excluded in spirit, though they land as ties in the aggregate counts. These should either be re-budgeted or dropped in future iterations.

Single model. All experiments use Claude Sonnet as the solver. Two replications are needed. First, a weaker model (e.g., Claude Haiku) would test the hypothesis that graph structure provides proportionally greater benefit to less capable models. Second, models from other vendors such as OpenAI's GPT or Google's Gemini, and open-weight families such as Llama, Qwen, DeepSeek, and Mistral would help to establish that the gains hold across LLMs rather than being specific to one model family.

5 RELATED WORK

Agent skills and their representations. ReAct [32] interleaves language-model reasoning with tool actions; Voyager [27] accumulates learned behaviors as code snippets indexed by prose; and Toolformer [21] teaches models to invoke external APIs mid-generation. Anthropic's Agent Skills [1, 3] standardize the packaging of such procedural knowledge, which a growing literature treats as institutional or expert knowledge to be transferred to agents [4] and surveys along axes of architecture and acquisition [30]. Closest to our work, SSL [14] also argues for structuring skill artifacts, but targets skill *discovery and assessment* rather than execution. These package procedural knowledge differently—as prose (Agent Skills), as code retrieved by prose descriptions (Voyager), or as structure aimed at skill discovery (SSL)—but none gives the skill a typed, schema-validated graph of scripted and prose steps, nor measures its effect on task execution; that is what AIP contributes.

Structured execution and workflow graphs. PAL [7] and Program of Thoughts [6] offload deterministic computation to code while reserving language-model reasoning for the rest, and chain-of-thought prompting [28] externalizes reasoning as explicit intermediate steps.

At the system level, GPTSwarm [35] casts language agents as optimizable computational graphs whose node prompts and edge connectivity are refined automatically by search or reinforcement learning—an early example of the agentic-computation-graph paradigm surveyed by Yue et al. [33], where the graph is the orchestration of a specific agent system, machine-optimized for that system. AIP instead places the graph at the level of a portable skill *artifact*, compiled from a human-authored skill and consumed by any general-purpose agent rather than bound to one bespoke system.

In practice, agent frameworks such as LangGraph and Google's ADK [8, 12] represent agent behavior as graphs of steps, DSPy [11] compiles declarative pipelines instead of hand-tuning prompts, and predefined workflow structure is both recommended [22] and benchmarked [29] as a route to reliability. AIP brings this graph view to the skill specification itself: scripted nodes carry deterministic work and prose nodes carry judgment, connected by typed input/output edges. These systems and AIP differ in kind, not degree (Table 3). LangGraph, ADK, and DSPy are runtime orchestration frameworks that an engineer programs in code to build a specific agent. AIP instead defines a portable skill *artifact* that a domain expert authors in prose and a compiler translates, consumed by a

general-purpose agent rather than executed by a bespoke runtime. Where those frameworks require authoring a graph from scratch, AIP compiles the large existing corpus of human-written prose skills into graph form, and—because it extends the Agent Skills specification—remains model- and vendor-agnostic with no runtime dependency. It also means that adding additional procedural knowledge comes at near-zero marginal cost, since AIP skills are auto-discovered and loaded on demand through progressive disclosure without any required changes to runtime orchestration code. DSPy is complementary but orthogonal in target: it optimizes the prompts or weights of a developer-defined pipeline, whereas AIP structures the skill representation itself.

This artifact-level, schema-validated typing is what enables the properties we rely on: node-addressable diagnosis and repair, with corpus-level governance queries and a bounded action space for learning over skills as future work.

Table 3: AIP vs. existing graph/workflow systems. AIP operates at the level of a portable, schema-validated *skill artifact* compiled from human prose, rather than engineer-written runtime orchestration.

	LangGraph/ ADK	DSPy	Agent Skills	AIP
Authored by	engineer (code)	engineer (code)	expert (prose)	expert +compiler
Source material	scratch	scratch	knowledge	existing skills
Portable artifact	no	no	yes	yes
Typed I/O edges	yes	partial	no	yes
Schema-validated	partial	no	no	yes
Corpus-queryable	no	no	no	yes

Benchmarking agentic systems. AgentBench [16] and SWE-bench [10] evaluate agents on real-world tasks with programmatic verifiers. SkillsBench, run through the BenchFlow SDK [5], instead isolates the incremental effect of a reusable skill by scoring each task with no skill, a curated skill, and a self-generated skill. We extend SkillsBench with AIP conditions to compare skill *formats* under a single solver.

Editing, self-improvement, and learning over skills. Editing a skill is unreliable for the reasons set out earlier—the content is unfamiliar and free-form prose offers no bounded surface to edit against, so model edits skew additive and intrinsic self-revision without feedback rarely helps [9, 23, 24]. A parallel line pursues agents that improve themselves [2, 18, 36], building on reinforcement learning from a reward signal [17, 25]. AIP's typed, schema-validated graph turns a skill edit into a bounded, checkable action and supplies the reward-bearing feedback loop these methods need at the level of an individual skill.

Governing skill corpora. As agentic systems scale, auditability and accountability become first-order concerns [20]. Since an AIP skill is a typed graph, a corpus of skills can be projected into a graph database [19] and queried for skills missing an approval step, shared sub-procedures, or reusable templates. This moves governance from manual documentation review to structured query and deterministic curation.

6 DISCUSSION AND FUTURE WORK

6.1 Experimental control

The highest-value missing experiment is an ablation separating the graph representation from agent scripting, in two arms. First, a *flattened* arm holds scripts fixed: each AIP-compiled package retains its `scripts/` and `references/` unchanged, with only the `SKILL.md` rewritten as plain-Markdown prose, so any delta against AIP measures the runtime value of the typed graph alone. Second, a *script-conversion* arm compiles scripts without the AIP spec, delivering the `SKILL.md` as a plain-Markdown Agent Skill rather than a graph. Because the AIP spec—with typed inputs and outputs—shapes script design, comparing this arm against the first measures the spec's authoring-time contribution. Together these disentangle graph representation from agent scripting alone. Our immediate future work will be to establish these arms as baselines, and confirm the promising behaviors observed in this paper.

6.2 From per-skill graphs to corpus governance

The same typed graph that makes a skill's deterministic steps runnable also makes the skill queryable. A library of AIP skills can be projected into a graph database, enabling audits such as identifying skills that lack an approval or validation step, discovering skills that share a common sub-procedure, and composing skills from reusable node templates. This moves skill governance from a manual documentation review to a structured query over a typed graph. An open question is whether providing an agent with query access to a skill corpus, rather than delivering a skill in-context as YAML, further improves task performance. We identify a controlled A/B experiment on this as a high-value next step.

6.3 Reinforcement learning over the skill graph

Left unconstrained, autonomous skill writing tends to accrete prose and code without pressure toward compression or toward the right boundary between scripted and natural-language nodes. The AIP execution graph provides a bounded, typed, validity-gated action space: edits are changes to nodes, scripts, or edges, each of which can be validated against the schema and evaluated against a reward signal. The manual revision cycle from v0.3a2 to v0.3a3—in which failures were diagnosed at the node level, corrections were made to the specification, and the updated skill was re-evaluated—is a policy step in this framework. Automating the edit–evaluate–edit loop constitutes reinforcement learning over skills [17, 25]. The node-level repair results of Section 4.4 serve as proof-of-concept that the feedback signal is both localizable and actionable. Formalizing this loop is a key topic for future work.

6.4 From specification to protocol

Today, AIP is closer to a specification than a protocol: the agent loads the entire YAML graph into its context window and follows the traversal logic through its own reasoning, with nothing enforcing adherence to the graph topology. This keeps AIP compatible with current agent-skill formats and lets us benchmark it, but it leaves headroom. Since traversal is unenforced, reliable execution still rests on the agent's own discipline, which is a ceiling that an

enforced protocol could raise, especially for smaller models. Loading the full specification into context on every run also adds a token burden that can raise cost and erode performance through context rot. We quantify this cost in Table 4: compiling to AIP increases the mean specification a task loads by roughly 78%, yet mean wall-clock *falls* from 585 s to 510 s (§4.3)—the added context does not make runs slower in practice.[3]

Since AIP already defines a typed action surface, a full protocol for walking the graph is feasible. It would execute nodes through controlled local or remote calls rather than in-context reasoning, while remaining backward-compatible with the agent-skill specification. We consider this the most important consideration for future work.

Table 4: Mean context-token and latency cost of AIP-compiled vs. human-curated skills, per task (n=27). SKILL.md is the specification loaded when a skill is invoked; references are loaded on demand. Compiling to AIP enlarges the loaded specification, yet end-to-end latency does not rise.

	Human	AIP	Δ
SKILL.md (tokens)	4,860	8,669	+3,809
references (tokens)	1,792	4,366	+2,574
wall-clock (s)	585	510	−75

7 CONCLUSION

Agent skills are written largely as free-form prose, resulting in the agent re-deriving procedures in every session that could have been captured with structure and code. This has a cost in terms of reliability and consistency on implementation-heavy tasks. Editing prose is something both humans and agents do poorly, for the distinct reasons set out in Section 1. AIP compiles a human-written skill into a directed execution graph: scripted where computation is deterministic, natural language where judgment is needed, connected by typed input/output edges and schema-validated throughout. Experts still author the skill; AIP makes its deterministic steps runnable and every step addressable.

The result is a representation that is more executable and improvable. Compiling human-written skills to AIP yields a statistically significant gain in task reward on SkillsBench, by handing the agent vetted, runnable units and a fixed procedure rather than asking it to re-plan from prose. Because each skill is a graph of named, typed, testable nodes, a failure localizes to a single node and is repaired by a checkable edit—a loop we ran by hand here, but one an adopter's agent can run directly. The same structure makes a skill corpus queryable for governance and provides a bounded, typed action space for reinforcement learning over skills.

AIP today is a specification an agent reads and follows; enforcing that traversal as a runtime protocol is a potential next step. But the core lesson already holds: a skill that is a graph is easier to execute, easier to diagnose when it fails, and easier to improve once diagnosed. The graph is not incidental to these properties—it is their common cause, and the foundation for skill libraries that can be reliably executed, governed, and learned.

[3]The clearest measure of cost in practice is the total number of tokens a task consumes end to end—dominated by the agent's multi-turn reasoning rather than the one-time specification. The SkillsBench harness does not log per-call token usage, so we report the loaded specification size and wall-clock here and leave full per-task token accounting to future work.

REFERENCES

[1] Agent Skills. 2025. *Agent Skills Specification.* Agent Skills. https://agentskills.io/specification

[2] Huan ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. 2025. A Survey of Self-Evolving Agents: What, When, How, and Where to Evolve on the Path to Artificial Super Intelligence. arXiv:2507.21046 [cs.AI] https://arxiv.org/abs/2507.21046

[3] Anthropic. 2025. *Equipping Agents for the Real World with Agent Skills.* Anthropic. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills Agent Skills open standard and reference SDK at https://agentskills.io.

[4] Gal Bakal. 2026. Knowledge Activation: AI Skills as the Institutional Knowledge Primitive for Agentic Software Development. arXiv:2603.14805 [cs.SE] https://arxiv.org/abs/2603.14805

[5] BenchFlow. 2026. *BenchFlow: Open-Source Benchmark Hub and Evaluation Infrastructure.* BenchFlow. https://docs.benchflow.ai/introduction

[6] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. arXiv:2211.12588 [cs.CL] https://arxiv.org/abs/2211.12588

[7] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided Language Models. arXiv:2211.10435 [cs.CL] https://arxiv.org/abs/2211.10435

[8] Google. 2025. *Agent Development Kit (ADK).* Google. https://adk.dev/ Open-source framework with sequential, parallel, and loop workflow agents.

[9] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large Language Models Cannot Self-Correct Reasoning Yet. arXiv:2310.01798 [cs.CL] https://arxiv.org/abs/2310.01798 ICLR 2024.

[10] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[11] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] https://arxiv.org/abs/2310.03714

[12] LangChain. 2025. *LangGraph.* LangChain, Inc. https://docs.langchain.com/langgraph Orchestration framework modeling stateful agent workflows as directed graphs of nodes and edges.

[13] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Binxu Li, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, Yizhuo He, Liqiang Jing, Luyang Kong, Xin Lan, Jiachen Li, Songlin Li, Yijiang Li, Yueqian Lin, Xinyi Liu, Xuanqing Liu, Haoran Lyu, Ze Ma, Bowei Wang, Runhui Wang, Tianyu Wang, Wengao Ye, Yue Zhang, Hanwen Xing, Yiqi Xue, Steven Dillmann, and Han chung Lee. 2026. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. arXiv:2602.12670 [cs.AI] https://arxiv.org/abs/2602.12670

[14] Qiliang Liang, Hansi Wang, Zhong Liang, and Yang Liu. 2026. From Skill Text to Skill Structure: The Scheduling-Structural-Logical Representation for Agent Skills. arXiv:2604.24026 [cs.AI] https://arxiv.org/abs/2604.24026

[15] Sherlock A. Licorish, Ansh Bajpai, Chetan Arora, Fanyu Wang, and Kla Tantithamthavorn. 2025. Comparing Human and LLM Generated Code: The Jury is Still Out! arXiv:2501.16857 [cs.SE] https://arxiv.org/abs/2501.16857

[16] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2025. AgentBench: Evaluating LLMs as Agents. arXiv:2308.03688 [cs.AI] https://arxiv.org/abs/2308.03688

[17] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL] https://arxiv.org/abs/2203.02155

[18] Maxime Robeyns, Martin Szummer, and Laurence Aitchison. 2025. A Self-Improving Coding Agent. arXiv:2504.15228 [cs.AI] https://arxiv.org/abs/2504.15228

[19] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph Databases: New Opportunities for Connected Data* (2 ed.). O'Reilly Media.

[20] Sandeep Saini. 2026. Governing the Agentic Enterprise: A New Operating Model for Autonomous AI at Scale. *California Management Review* (2026). https://cmr.berkeley.edu/2026/03/governing-the-agentic-enterprise-a-new-operating-model-for-autonomous-ai-at-scale/ Published online March 20, 2026.

[21] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761 [cs.CL] https://arxiv.org/abs/2302.04761

[22] Erik Schluntz and Barry Zhang. 2024. *Building Effective Agents*. Anthropic. https://www.anthropic.com/engineering/building-effective-agents

[23] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2024. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I Learned to Start Worrying About Prompt Formatting. arXiv:2310.11324 [cs.CL] https://arxiv.org/abs/2310.11324 ICLR 2024.

[24] Prasann Singhal, Tanya Goyal, Jiacheng Xu, and Greg Durrett. 2024. A Long Way to Go: Investigating Length Correlations in RLHF. arXiv:2310.03716 [cs.CL] https://arxiv.org/abs/2310.03716 COLM 2024.

[25] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2 ed.). MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[26] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C.J. Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[27] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291 [cs.AI] https://arxiv.org/abs/2305.16291

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Vol. 35. 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[29] Ruixuan Xiao, Wentao Ma, Ke Wang, Yuchuan Wu, Junbo Zhao, Haobo Wang, Fei Huang, and Yongbin Li. 2024. FlowBench: Revisiting and Benchmarking Workflow-Guided Planning for LLM-based Agents. arXiv:2406.14884 [cs.CL] https://arxiv.org/abs/2406.14884

[30] Renjun Xu and Yang Yan. 2026. Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward. arXiv:2602.12430 [cs.AI] https://arxiv.org/abs/2602.12430

[31] Wenda Xu, Guanglei Zhu, Xuandong Zhao, Liangming Pan, Lei Li, and William Yang Wang. 2024. Pride and Prejudice: LLM Amplifies Self-Bias in Self-Refinement. arXiv:2402.11436 [cs.CL] https://arxiv.org/abs/2402.11436

[32] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[33] Ling Yue, Kushal Raj Bhandari, Ching-Yun Ko, Dhaval Patel, Shuxin Lin, Nianjun Zhou, Jianxi Gao, Pin-Yu Chen, and Shaowu Pan. 2026. From Static Templates to Dynamic Runtime Graphs: A Survey of Workflow Optimization for LLM Agents. arXiv:2603.22386 [cs.AI] https://arxiv.org/abs/2603.22386

[34] Yusen Zhang, Sarkar Snigdha Sarathi Das, and Rui Zhang. 2024. Verbosity ≠ Veracity: Demystify Verbosity Compensation Behavior of Large Language Models. arXiv:2411.07858 [cs.CL] https://arxiv.org/abs/2411.07858

[35] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Language Agents as Optimizable Graphs. In *International Conference on Machine Learning (ICML)*. arXiv:2402.16823 [cs.AI] https://arxiv.org/abs/2402.16823

[36] Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. 2025. Self-Adapting Language Models. arXiv:2506.10943 [cs.LG] https://arxiv.org/abs/2506.10943 NeurIPS 2025.