Lumos: One Skill, One Graph, Many Queries

An Agent-Skill Design Pattern for Cross-Modal Artifact Understanding

Jacqueline Pang
pangjacque@outlook.com
Independent Researcher
United States

Abstract

Modern machine-learning projects fragment knowledge across three artifact classes: production code authored by engineers, exploratory Jupyter notebooks authored by data scientists, and modeling governance documents authored by reviewers and auditors. IDEs reason about each modality in isolation, leaving cross-artifact lineage—which production class is exercised by a Jupyter cell, which Python wrapper crosses into a JVM runtime—inaccessible. We present *Lumos*, an Agent Skill that composes seven specialist agents and three user-facing skills into a cross-modal knowledge-graph pipeline. The contribution is the design pattern: an *orchestrator skill* dispatches *specialist agents* in parallel, a *cross-modal resolver* links entities across modalities, a *graph reviewer* enforces structural hygiene, and downstream *consumer skills* query a shared JSON artifact. This decoupling makes the pattern a reusable template for packaging procedural knowledge in domains where artifacts cross format and runtime boundaries. We instantiate Lumos for source code (Python and Scala in the current implementation; the per-language specialist design extends to R, Julia, and other ecosystems), Jupyter notebooks (with magic-aware extraction), Markdown/PDF documentation, and Python→JVM Py4J cross-runtime bridges. We report a feasibility case study on FLAML—2,155 nodes, 4,373 edges, 178 *import-grounded* cross-modal edges (notebook↔code imports and calls plus Python→JVM Py4J bridges)—and a small hand-labeled precision study (N=60) reporting 83.7% precision on the import-grounded regime versus an explicitly preliminary 23.5% on document-bound edges.

Keywords: Agent Skills, Procedural Knowledge, Knowledge Graph, Cross-modal Lineage, Multi-modal artifacts, Static Analysis Skills, Skill Composition

1 Introduction

Quantitative-analysis work—machine learning, data analysis, data engineering, model governance—spans three communities that author three different artifact classes. Engineers write production code (Python, Scala, etc.). Data scientists and analysts explore in Jupyter notebooks. Model-validation teams write audit notes, data dictionaries, and governance reviews. Each artifact references the others, but no commonly available tool resolves those cross-modal references. Standard IDE features ("find all references," "go to definition") use language-specific servers that do not "talk" to each other; they are silent on questions like *"which function in code base is exercised by this jupyter cell?"*, *"which notebook computes the Gini metric defined in this governance document?"*, or *"which Scala UDF is invoked by this PySpark notebook through Py4J?"*.

Agent Skills [1] are a promising substrate for closing this gap. A Skill can encode the procedural knowledge needed to walk three distinct artifact graphs, stitch them together, and expose the result to downstream skills for natural-language Q&A and pre-commit impact analysis. The design problem is concrete: the pipeline must be *build-once-query-many* (one run produces a fixed graph that consumer skills query without re-execution), *script-heavy* rather than markdown-heavy (logic lives in `bin/*.py` rather than agent prompts, so per-query token cost stays bounded), and *harness-portable*.

This paper presents *Lumos*, a Skill composition meeting these constraints. *The central contribution is a Skill packaging pattern, not a new static-analysis algorithm: deterministic analyzers produce typed intermediate artifacts, a resolver owns cross-modal semantics, and consumer skills query a stable graph without re-running producers.* Concretely, the orchestrator is `skills/everything/SKILL.md`, specialist roles live as `agents/*.md`, deterministic logic lives in `bin/*.py`, and the produced artifact is `.lumos/knowledge-graph.json`. The same bundle targets multiple agent harnesses through plugin manifests—Claude Code, GitHub Copilot, and (in progress) Cursor; the complete implementation (bundle, scripts, schema, FLAML graph, viewer, prompts, and evaluation labels) is released under MIT at https://github.com/pangjacque/Lumos. This work primarily targets the *Agent Skills & Applications* workshop theme and contributes preliminary graph-derived structural metrics relevant to *Evaluation & Benchmarking*.

Why a Skill, not a static-analysis tool. The deterministic scripts could ship as a stand-alone CLI; the Skill substrate adds three jobs a tool cannot: orchestration (dispatching specialists, recovering from per-specialist failures), classification and natural-language steps that defy regular grammar (document-class assignment, Q&A in `/lumos:chat`, intent-aware diff narration), and a documented composition boundary so third-party consumers compose without forking producers.

Contributions.

(C1) A **Skill design pattern** for cross-modal artifact understanding: *orchestrator-skill → specialist-agent fan-out → cross-modal resolver → graph reviewer → shared JSON artifact → consumer skills* (Sec. 3).
(C2) **Lumos**, an instantiation of the pattern for ML project intelligence, packaged as a single Skill bundle that runs across multiple agent harnesses (Sec. 4).
(C3) An **edge-typed cross-modal graph schema** with first-class `cross_boundary_*`, `cell_data_flow`, and `py4j_*` edges (Sec. 5).
(C4) A **preliminary case study on FLAML** reporting structural metrics and a small (N=60) hand-labeled precision study with named failure modes, separating an import-grounded regime (83.7% precision) from an explicitly preliminary document-bound regime (23.5%) (Sec. 6).

2 Related Work

Skills as first-class artifacts. Anthropic introduced Agent Skills as composable folders of instructions, scripts, and resources keyed by a `SKILL.md` file [1]. Recent surveys situate Skills along architecture, acquisition, deployment, and security axes [7]. Empirical studies of public Skills marketplaces find that the population is dominated by short, single-purpose Skills with relatively shallow procedural content [3]. SkillsBench [2] provides the first task-grounded evaluation: across 86 tasks in 11 domains, *curated* Skills raise average pass rate by 16.2 percentage points, while *self-generated* Skills offer no net benefit. SkillsBench evaluates Skills as task-level interventions, and*Lumos is complementary, proposing a reusable Skill architecture for producing cross-artifact knowledge graph whose output is itself an evaluable artifact.* These results motivate hand-written, domain-specific Skill design with deterministic scripts and explicit validation—the regime in which Lumos sits.

Skill safety. Adjacent work studies the security surface of the Skills ecosystem. The Snyk *ToxicSkills* audit found that 1,467 of 3,984 audited Skills (36.8%) contained at least one security flaw, with 534 rated critical [5]; an academic empirical study reports a similar pattern at scale [4]. We treat these findings as a constraint on Lumos's design: producers are read-only and outputs are validated by an explicit reviewer (Sec. 4); we discuss the residual artifact-level risk in Sec. 7.

Code-understanding tooling. Existing tools attack subsets of the cross-modal lineage problem. Table 1 sketches the *capability gap*—a qualitative coverage matrix, not a quantitative benchmark—against four common alternatives. Lumos's contribution is not any single capability in isolation—`grep`, import graphs, and language servers each do part of the job—but a single persisted artifact that lets consumer skills query all four dimensions through one interface.

Table 1. Qualitative *capability comparison* (not a quantitative benchmark) on a project bundling Python, Jupyter, Scala via Py4J, and Markdown. ✓ = supported; ∼ = partial or setup-dependent; × = unsupported.

Method	nb→code call	nb→code import	Py4J bridge	Reusable graph
`grep`/`rg` on names	∼	∼	∼	×
Python import graph (`pydeps`)	×	✓	×	∼
LSP *find references*	∼	∼	×	×
Lumos	✓	✓	✓	✓

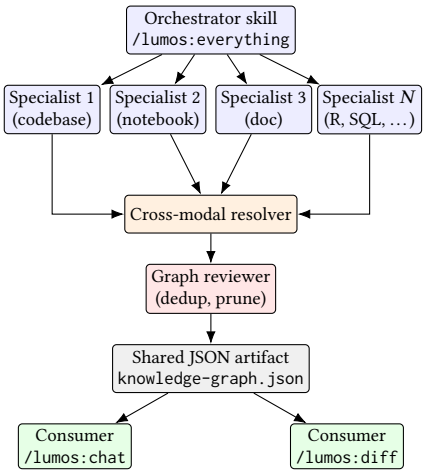

Figure 1. The Skill design pattern. An *orchestrator* fans out to specialist agents whose per-modality outputs are linked by a *resolver*, validated by a *reviewer*, and emitted as a shared artifact. Stateless *consumer skills* query the artifact without re-running production.

Pipeline-lineage systems (MLflow, OpenLineage) are complementary: they track *run-time* lineage (datasets, model artifacts) rather than *authoring-time* lineage between source artifacts.

3 The Design Pattern

We abstract the pattern that Lumos instantiates because we believe it transfers to any domain where (i) procedural knowledge must process artifacts in multiple formats or runtimes, and (ii) downstream agents need a stable, queryable representation of the result. Concretely, the pattern requires six roles: an orchestrator that fans work out, per-modality specialists producing typed intermediate artifacts, a resolver that owns *all* inter-modal edges, a reviewer enforcing structural invariants, a shared on-disk artifact as the producer/consumer boundary, and stateless consumers that query it without re-execution. Folder layout and prompt/script separation are implementation conveniences; these six roles are what reuse depends on.

Authoring principles. Two constraints govern every Lumos `SKILL.md`. *Thin instructions, fat scripts*: the `SKILL.md` describes *when* and *what* to invoke; procedural details (AST traversal, Jupyter-magic regex, Py4J pattern matching) live in deterministic Python scripts under `bin/`, keeping each agent's context window small and the procedural knowledge unit-testable. *Specialists do not communicate*: every specialist writes a self-contained intermediate JSON, making them embarrassingly parallel and re-runnable. Together these decouple producers from consumers via the artifact—a new consumer (e.g. a governance-report generator) requires no change to producers, enabling third-party Skill composition without forking the pipeline.

4 Lumos: An Instantiation

Lumos realises the pattern for ML project intelligence. The orchestrator is `skills/everything/SKILL.md`, specialists live as `agents/*-analyzer.md`, deterministic logic lives in `bin/*.py`, and the artifact is `.lumos/knowledge-graph.json`. Lumos packages seven specialist agents (four source-extraction analyzers for code, notebooks, Scala, and docs; a project scanner; the resolver; the reviewer) and three user-facing skills (`/lumos:everything`, `/lumos:chat`, `/lumos:diff`). Per-harness adaptation is limited to a single manifest file (e.g. `.claude-plugin/plugin.json`); skill bodies, agents, and `bin/` scripts are reused verbatim across Claude Code, GitHub Copilot, and (in progress) Cursor.

Specialists. Four specialists handle source extraction. The *code analyzer* (`agents/code-analyzer.md` + `bin/extract_code.py`) walks the Python AST to emit `file`, `function`, `class` nodes and `contains`/`imports` edges. The *notebook analyzer* (`bin/extract_notebook.py`) parses `nbformat` cells with one critical wrinkle: Jupyter magics (`%%time`, `%%sql`, `%run`, `!pip install ...`) are syntactically invalid Python, so they must be stripped *before* AST parsing. Lumos strips them with a regex preprocessor and *preserves them as cell metadata*, distinguishing (e.g.) `%%sql` cells from Python cells in the resulting graph. The same specialist tracks per-cell variable definitions and uses to emit `cell_data_flow` edges and detects execution anomalies (out-of-order, error, never-executed cells). The *Scala analyzer* (`bin/extract_scala.py`) uses `tree-sitter` to extract case classes, traits, and Spark UDF patterns. The *doc analyzer* (`bin/extract_doc.py`) uses `markitdown` over Markdown, PDF, and Word documents and classifies each into {`model_doc`, `data_dictionary`, `governance`, `design_doc`} via keyword scoring; an experimental sub-module attempts to extract named metric tokens, but as Sec. 6 shows, this channel is fragile and we do not claim it as production-ready. A project scanner (`bin/scan_project.py`) provides file inventory. New languages enter as new specialists: an R or Julia analyzer would add a single `bin/extract_<lang>.py` plus an `agents/<lang>-analyzer.md`, with no change to the resolver, reviewer, or consumer skills.

Table 2. Lumos edge schema, with the producer responsible for emitting each type. CM = cross-modal.

Group	Edge type	Producer
Containment	`contains`	all specialists
Intra-code	`imports`	code analyzer
	`inherits`	code analyzer
Intra-notebook	`cell_flow`	notebook analyzer
	`cell_data_flow`	notebook analyzer
CM (nb ↔ code)	`cross_boundary_import`	resolver
	`cross_boundary_call`	resolver
CM (doc ↔ code/nb)	`doc_references_code`	resolver
	`doc_references_notebook`	resolver
	`defines_metric`	doc analyzer
	`metric_implemented_by`	resolver
Cross-runtime	`py4j_bridge`	resolver
	`py4j_method_call`	resolver

Cross-modal resolver. `bin/resolve_imports.py` is the only Lumos component that crosses modality boundaries. It builds a *module map* (file path ↦ Python module path), an *entity map* (function/class name ↦ node ID), and a *path map* for doc reference resolution. For each notebook `import` or `from` M `import` N, it tries (i) the full module-qualified name, (ii) the bare symbol, and (iii) Python's `importlib.util.find_spec` as a fallback to detect whether the target is local or third-party. Local matches become `cross_boundary_import` or `cross_boundary_call` edges; non-local references are dropped. The resolver also pattern-matches Py4J bridges (`spark._jvm.<package>.<Class>`) and emits `py4j_bridge` / `py4j_method_call` edges. Document-to-code linking is currently a heuristic name match between extracted metric tokens and code symbols and is the noisiest cross-modal channel (Sec. 6).

Reviewer and consumers. The reviewer (`bin/merge_graph.py`) deduplicates nodes by `id` and edges by `(source, target, type)` and prunes dangling edges. `skills/chat/SKILL.md` answers natural-language questions strictly over the JSON artifact ("which notebooks use `X`?", "what breaks if I change `Y`?"); `skills/diff/SKILL.md` traverses the graph 1–2 hops from each file in `git diff` to produce a pre-commit blast-radius report. Both consumers are stateless functions of the artifact.

5 Graph Schema

The Lumos schema has 8 node types (`file`, `class`, `function`, `notebook`, `cell`, `document`, `doc_section`, `metric_def`) and the 13 edge types listed in Table 2. We organise edges into six groups so that schema users (downstream consumer skills) can reason about which subgraphs to traverse for a given query.

Table 3. FLAML knowledge-graph composition. Cross-modal edges are produced by the resolver and span format or runtime boundaries.

Node type	Count	Edge type	Count
function	981	contains	2,021
cell	833	cell_flow	797
class	149	imports	711
file	91	cell_data_flow	514
doc_section	58	inherits	115
notebook	36	**cross_boundary_import**	**111**
document	6	**cross_boundary_call**	**64**
metric_def	1	**metric_implemented_by**	**20**
		doc_references_code	**10**
		doc_references_notebook	**6**
		py4j_bridge	**2**
		py4j_method_call	**1**
		defines_metric	**1**
total	**2,155**	**total**	**4,373**

The most useful design observation is that *every cross-modal edge type is produced by the resolver*, while every intra-modal type comes from a single specialist. This locality keeps specialists independent and makes the resolver the natural place to add new cross-modal capabilities (e.g., notebook-to-doc edges, or cross-language calls beyond Py4J) without modifying the producers.

Artifact shape. A representative node and edge:

```
{"id":"func:AutoML.fit","type":"function",
 "file":"flaml/automl.py","line":127}
{"source":"cell:demo.ipynb#3","target":"func:AutoML.fit",
 "type":"cross_boundary_call","evidence":"AutoML().fit(X,y)"}
```

Every edge carries an `evidence` string that consumers resolve back to source for citation.

6 Preliminary Case Study: FLAML

We exercise Lumos on FLAML [6][1], a publicly available AutoML library that bundles a production Python codebase, an example notebook gallery, a small Spark/Scala demo subproject, and Markdown documentation. We report structural metrics for the produced graph and a small hand-labeled precision study on the import-grounded regime.

Setup. We invoke `/lumos:everything` on the FLAML repo. The project scanner inventoried 162 files: 91 code (90 Python, 1 Scala), 36 Jupyter notebooks, 6 Markdown documents, 27 configuration files, and 2 other. The pipeline runs end-to-end in roughly a minute on a contemporary laptop.

Structural / coverage metrics. Table 3 reports the resulting graph. 178 of 4,373 edges (4.1%) cross a modality or runtime boundary via the import resolver and Py4J matcher (the *import-grounded* regime); a further 37 doc-bound edges rely on name-token matching (the *exploratory* regime).

Manual precision study. On a 60-edge stratified random sample (43 of 178 import-grounded plus 17 of 37 exploratory, oversampling the smaller bucket; seed 42; rubric in supplementary `eval/`), we hand-labeled each as `correct`, `wrong-target`, `false-edge`, or `ambiguous`. The two regimes separate sharply (Wilson 95% CIs):

- *Import-grounded* (`cross_boundary_*`, `py4j_*`): **36/43 = 83.7%** (70.0%–91.9%). Per-type: `cross_boundary_call` 19/20 (95.0%), `cross_boundary_import` 14/20 (70.0%), `py4j_*` 3/3 (100%).
- *Exploratory* (`doc_references_code`, `metric_implemented_by`): **4/17 = 23.5%** (9.6%–47.3%).

Every failure clusters on a small number of named, fixable bugs. Import-grounded misses: 5 `from package import name` resolutions stop at the package's `__init__.py` instead of following re-exports; 2 are third-party `prophet` name collisions matched against an internally-named class. Exploratory misses: 5/10 `doc_references_code` edges have `NULL` target nodes (a malformed-edge bug); 7/7 sampled `metric_implemented_by` edges originate from a single mis-extracted token (`roc=60`, a numeric time-budget read as a metric) propagated via substring matching. We treat the import-grounded regime as feasibility evidence and the exploratory regime as documenting two named resolver bugs, not as ready-for-use signals.

The 3 `py4j_*` edges link a Python wrapper to two Scala classes (`MetricsCalculator`, `ModelEvaluator`) in the bundled `spark_demo/` subproject—a Python→JVM dependency Python-only import analysis would miss.

Consumer-skill grounding. Figure 2 shows the same artifact rendered in a browser-based viewer answering one such query. Three representative queries illustrate the pattern: (a) *"Which notebooks call* `AutoML`*?"*—`cross_boundary_call` edges whose target matches the `AutoML` class (4 cells, 4 notebooks); (b) *"What is affected if* `sklearn_metric_loss_score` *changes?"*—a 1–2-hop traversal yielding 38 cells (Fig. 2); (c) *"Where does Python cross into Scala?"*—direct enumeration of `py4j_*` edges. Each answer reduces to a graph query whose evidence is the supporting edge IDs.

7 Discussion, Limitations, and Future Work

Where the pattern transfers. This six-role pattern fits any domain where procedural knowledge must process artifacts in multiple formats or runtimes and multiple consumers want different views of one underlying graph. Beyond ML: infrastructure code with sibling configuration and runbooks; scientific data with sibling pipelines and protocol papers. In each, the resolver carries the cross-modal procedural knowledge that no single specialist could embed, and consumers (governance dashboards, audit reports, dependency visualisations) can be added without forking or

[1] https://github.com/microsoft/FLAML

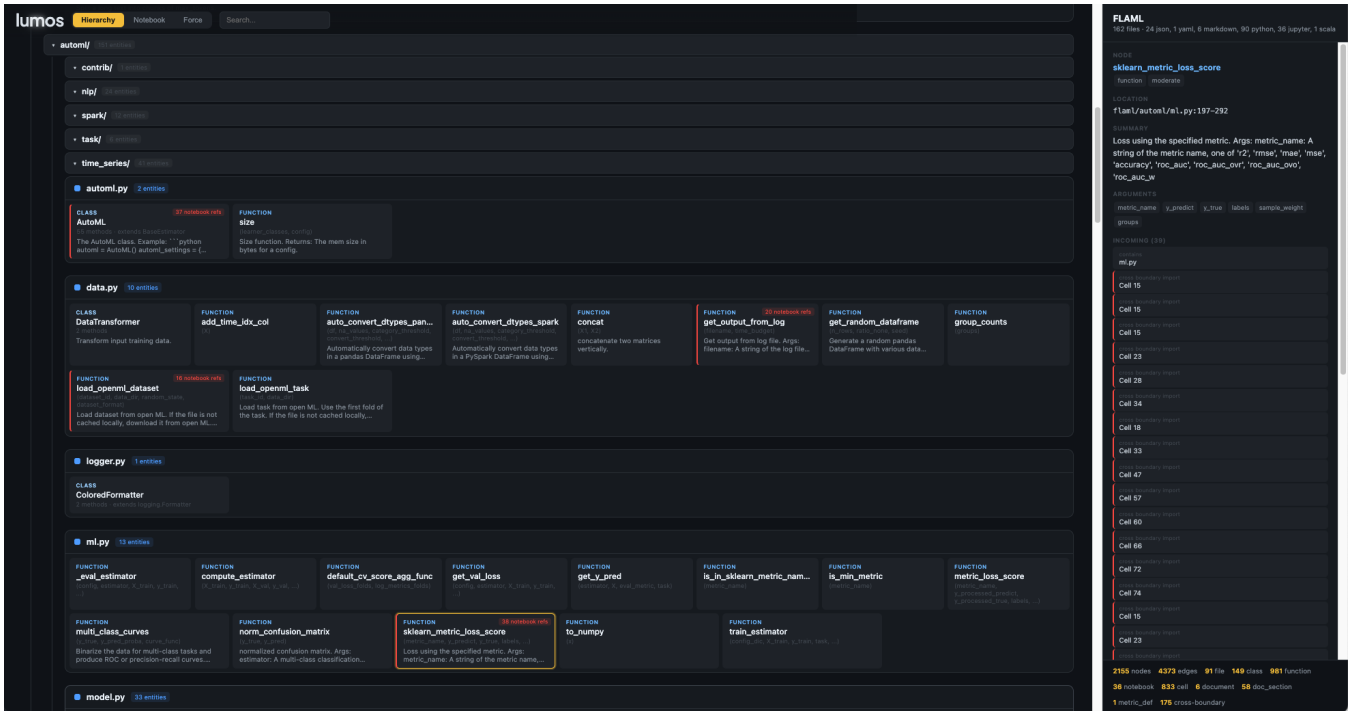

Figure 2. Browser view of the Lumos artifact for FLAML, answering query (b). Selecting `sklearn_metric_loss_score` highlights the function and the 38 notebook cells reaching it via `cross_boundary_call` and `cross_boundary_import` edges. The 38-cell count is graph-traversal output; per-edge precision for the import-grounded regime is reported in §6 (36/43 = 83.7%).

invoking the producers—an underused leverage point in an ecosystem where most Skills are monolithic.

Safety surface. Lumos producers are read-only over the project tree; the *emitted artifact* is a downstream supply-chain risk surface, since a buggy or compromised specialist could inject fabricated edges, false `evidence`, or mistyped edges that bypass filters. The reviewer enforces three invariants (no dangling references, edge-tuple deduplication, node-id uniqueness), and `/lumos:chat` resolves every cited edge to its evidence file before answering, so fabricated edges surface as unverifiable citations. Given the 36.8% flaw rate in the recent 3,984-Skill audit [5], signing the artifact and adding consumer-side schema validation are natural next steps.

Roadmap. A larger adjudicated precision study with two annotators and inter-rater agreement; the two named resolver fixes identified by the FLAML sample (re-export following on `from package import name`; third-party-collision filtering for shared symbol names); a multi-project corpus including PySpark, R, and Julia content; integration with SkillsBench-style task evaluation [2]; specialists for R, Julia, and SQL; and incremental-rebuild support.

8 Conclusion

We presented *Lumos*, an Agent Skill composition for cross-modal knowledge-graph construction over code, notebooks, and documentation, and abstracted the design pattern that underlies it. The FLAML case study demonstrates feasibility, not benchmark superiority: 2,155 nodes and 4,373 edges including 178 import-grounded cross-modal edges achieving 83.7% per-edge precision on a 43-edge sample (95.0% for calls, 100% for Py4J), with the document-bound regime documenting two named resolver bugs rather than being declared ready. We invite the community to apply the same pattern to adjacent multi-artifact domains and to share the evaluation harnesses needed to compare cross-modal Skills as first-class artifacts.

References

[1] Anthropic. 2025. Introducing Agent Skills. Anthropic news post. https://www.anthropic.com/news/skills The SKILL.md specification was opened as a community standard in December 2025..

[2] 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 preprint arXiv:2602.12670* (Feb. 2026). https://arxiv.org/abs/2602.12670

[3] George Ling, Shanshan Zhong, and Richard Huang. 2026. Agent Skills: A Data-Driven Analysis of Claude Skills for Extending Large Language Model Functionality. *arXiv preprint arXiv:2602.08004* (Feb. 2026). https://arxiv.org/abs/2602.08004

[4] Yi Liu, Weizhe Wang, Ruitao Feng, Yao Zhang, Guangquan Xu, Gelei Deng, Yuekang Li, and Leo Zhang. 2026. Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale. *arXiv preprint arXiv:2601.10338* (Jan. 2026). https://arxiv.org/abs/2601.10338

[5] Snyk Security Research. 2026. ToxicSkills: A Supply-Chain Audit of 3,984 Agent Skills. Snyk blog. https://snyk.io/blog/toxicskills-malicious-ai-agent-skills-clawhub/#toxic-skills-summary 1,467 of 3,984 audited Skills (36.82%) contained at least one security flaw; 534 were rated critical..

[6] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. 2021. FLAML: A Fast and Lightweight AutoML Library. In *Proceedings of Machine Learning and Systems (MLSys).* https://github.com/microsoft/FLAML

[7] Renjun Xu and Yang Yan. 2026. Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward. *arXiv preprint arXiv:2602.12430* (Feb. 2026). https://arxiv.org/abs/2602.12430