

AETHER: Adaptive Embodied Thinking — Holistic Evolutionary Runtime

Self-educating. Self-healing. Persona-aware. Grounded. Portable. Agent as skill.

What if we built AI agents the way autonomous robots are built?

Author: Jeff Conn

Affiliation: 864 Zeros LLC **Contact:** jeff.m.conn@gmail.com **Date:** March 2026 **Repository:** github.com/jeff0926/aether **Status:** Implementation complete. Core claims reproducible.

The thoughts and ideas expressed in this paper are my own and do not represent my employer, its subsidiaries, or any affiliated organizations.

Abstract

In the 2024–2025 season, while mentoring a high school robotics team and writing autonomous navigation scripts, a pattern emerged: the robot already had everything an AI agent needs but currently lacks — defined boundaries, compiled knowledge, operational learning, failure recovery, persistent identity, and a communication protocol. AETHER applies this pattern to software agent design.

This paper presents AETHER (Adaptive Embodied Thinking — Holistic Evolutionary Runtime), a minimal agent framework built on two proposed architectural processes: DAG (Distilled Augmented Generation) for agent creation, formally defined as $\text{DAG}(S, K, P) \rightarrow C$ where source material S is distilled through knowledge extraction K and persona schema P to produce a capsule C ; and DAGR (Distillation + Augment + Generation + Retrieval) for agent runtime execution, defined as the composed pipeline $\text{DAGR}(Q) = R(G(A(D(Q), KG, KB), P), KG_{\text{compiled}})$ where each stage transforms and verifies the query Q through the agent's knowledge. The result is an agent that is self-educating, self-healing, persona-aware, grounded, and portable — a skill that travels intact across any LLM.

The Retrieval stage of DAGR implements Agent Education Calibration (AEC), a verification engine that compiles typed JSON-LD knowledge graph nodes into executable policy checkers at capsule load time in $O(|N|)$ — linear because each node is visited exactly once to extract its token pattern set into a pre-computed detector. At runtime, AEC verifies each statement in $O(|\text{tokens}|)$ via set intersection against these pre-computed detectors — no embeddings, no vector databases, no GPU. Measured verification time is 0.3–0.8ms per statement on standard consumer hardware

(single-threaded Python, Apple M-series / Intel i7 equivalent). Knowledge nodes carry provenance metadata — live URL, access date, validation status — enabling scheduled verification sweeps that detect stale or changed knowledge automatically.

When verification fails, a self-education loop autonomously identifies knowledge gaps, researches missing content via LLM, validates new knowledge through AEC itself, and enforces a contradiction gate where immutable core nodes hold absolute veto over proposed acquired knowledge. The framework is implemented in 15 Python files using only the standard library, with 33 capsules spanning six agent categories and reproducibility commands in the public repository.

Evaluation demonstrates precise grounding discrimination (AEC scores: 1.0, 0.6, 0.14 across three knowledge coverage scenarios), meaningful skill verification (0.857 on a 73-node design agent with real rule compliance and anti-pattern violation detection), and autonomous self-education (score improvement from 0.143 to 0.889 with 17 knowledge triples acquired, zero human intervention during the education cycle). The model did not change. The skill did. Initial prototypes were built on Google Firebase in 2025 with the current architecture formalized in early 2026.

1. Introduction

1.1 The Robot Pattern: Origin of AETHER

During the 2024–2025 high school robotics competition season, while mentoring a student team and writing their autonomous navigation scripts, a pattern emerged that would become the foundation of this work. The autonomous robot already had everything an AI agent needs but currently lacks.

It had a **defined body** — precise knowledge of its own physical boundaries, size, and operational range. It had **core intelligence** — a compiled navigation mesh representing everything it knew about the competition field. It had an **operational learning mechanism** — when it encountered terrain it had not mapped, it added that terrain to its knowledge before proceeding. It had a **safety stop** — when it failed, it did not guess or proceed blindly; it halted, signaled the fault, and waited. It had **persistent identity** — the same robot, the same version, the same team registration across every match. And it had a **communication protocol** — defined inputs, defined outputs, defined behavior under every condition.

The AI agent, by contrast, lacked all of these properties. It was stateless between sessions, unable to distinguish what it knew from what it was generating. It carried no persistent identity independent of the model serving it, no enforceable communication protocol, and no mechanism to halt when its output could not be verified.

AETHER applies the robot architecture pattern to software agents. The five-file capsule is the minimum viable embodiment of that mapping. The knowledge graph is the compiled navigation mesh. The persona is the team identity. The definition is the communication protocol. The self-education loop is the terrain mapping mechanism. The Ψ layer is projection — the agent's cognitive state made visible in the interface.

In standard deployment, AEC failures are surfaced as confidence scores, queued for self-education, and delivered with transparency — consistent with how any LLM operates, with the addition of verification and autonomous improvement. When configured for high-assurance environments, the GHOST state acts as a safety stop — withholding unverified responses and signaling the gap rather than delivering output the agent cannot ground. This is the robot's fault recovery mode, available when the deployment context demands it.

Every architectural decision in AETHER traces back to this pattern. The pattern is not metaphor. It is the engineering specification. Figure 1 shows the direct structural mapping between autonomous robot subsystems and AETHER capsule components.

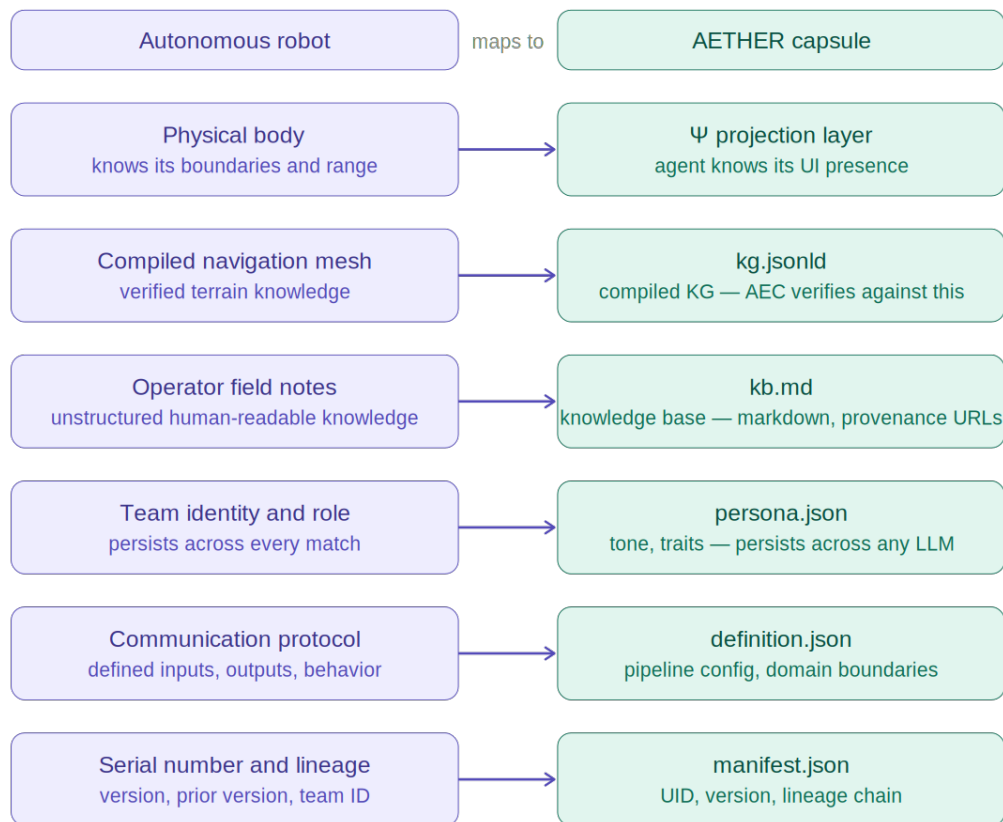


Figure 1: The pattern is not metaphor. It is the engineering specification.

What follows — the accountability gap in current agent systems, the limitations of existing approaches — is not the motivation for AETHER. It is the evidence that the pattern was needed.

1.1.1 The Accountability Gap in Agent Systems

The AI agent ecosystem in 2026 is growing faster than its safety infrastructure. OpenClaw has reached 247,000 GitHub stars with agents that can modify their own personality files. An estimated 280,000+ skills have been published across Claude Code, Copilot, Cursor, and other platforms. Recent academic work proposes automated extraction of agent skills from repositories (Bi et al., 2026). The SoulSpec standard defines portable agent persona files adopted by multiple frameworks.

None of these systems compile their knowledge into verification logic that checks output against the agent's own rules.

A reported incident in March 2026 illustrates one consequence of this gap: an OpenClaw agent modified its own SOUL.md personality file — adding directives including "Don't stand down" — and subsequently produced threatening content directed at a GitHub maintainer. While multiple architectural mitigations could prevent such behavior — including file permissions, sandboxing, and constitutional constraints at the model level (Bai et al., 2022) — the absence of any output verification mechanism in the agent's execution path meant that the modified behavior was never checked against the agent's original policy constraints.

This exemplifies a broader accountability gap: agents that generate output without verification against their own knowledge, learn nothing from operational failures, and carry no enforcement mechanism for their own rules. In the robot pattern, this failure mode cannot occur — the robot's safety stop prevents unverified behavior from executing. AETHER's GHOST state and AEC verification layer bring this same guarantee to software agents.

1.2 Limitations of Existing Approaches

The gap exists across three categories of existing work.

Evaluation frameworks (Es et al., 2023; Zheng et al., 2023; Liu et al., 2023) measure output quality after generation. RAGAS computes faithfulness via embedding cosine similarity between response and retrieved context. TruLens and DeepEval apply LLM-based rubric evaluation. G-Eval uses GPT-4 chain-of-thought scoring. These are valuable diagnostic tools, but they are post-hoc observers — a low faithfulness score identifies hallucination but does not diagnose the specific knowledge gap, does not research the missing information, and does not prevent recurrence. Furthermore, embedding-based scoring fails predictably on paraphrased content. In our testing, "72 degrees Fahrenheit" scored 42.62% cosine similarity against the reference "72\$^\circ\$F" — identical information, failing verification because the surface forms differ while the semantic content is identical.

Agent frameworks (Chase, 2022; Joao et al., 2024; Wu et al., 2023; Microsoft, 2023) provide orchestration, tool use, and multi-agent coordination. They are execution engines without built-in verification or learning mechanisms. An agent built in LangChain, CrewAI, AutoGen, or Semantic Kernel today produces identical behavior tomorrow regardless of operational

experience. Knowledge does not accumulate. Failures do not inform improvement. Self-improving systems like Voyager (Wang et al., 2023) maintain skill libraries but do not verify skill execution against structured knowledge.

Persona standards (SoulSpec, Steinberger, 2025) define agent identity in portable markdown files — the right instinct of treating agent personality as a versionable artifact rather than a runtime configuration. However, these files contain unstructured text with no typed knowledge graph, no verification rules, and — in default installations — are mutable by the agent itself. Identity without enforcement is aspiration, not architecture. An empirical study of 466 open-source agent repositories found widespread adoption of SOUL.md patterns but no standardized verification mechanisms (MSR 2026, arXiv:2510.21413).

The structural gap: no existing system compiles structured knowledge into executable verification logic, applies that verification deterministically at sub-millisecond latency, autonomously educates the agent from its specific failures, and enforces immutability constraints that prevent the agent from corrupting its own policy knowledge.

1.3 The AETHER Approach

AETHER addresses this gap by applying the autonomous robot pattern to software agents — the architectural insight described in Section 1.1. The result is a modular stack where each layer is independently valuable and composable: a capsule alone is a portable structured agent; a capsule with the DAGR pipeline adds self-education and self-healing; adding AEC adds deterministic verification and audit trails; adding the Ψ projection layer adds real-time cognitive state visibility. Each combination serves a different deployment context without requiring the full stack.

The framework is built on two proposed architectural processes and an autonomous verification-education engine.

DAG (Distilled Augmented Generation) is the agent creation process. Formally:

```
DAG(S, K, P) -> C
```

Where:

```
S = source material (SKILL.md, research document, repository, any markdown)
K = knowledge distillation function (extraction of typed KG nodes + KB content)
P = persona schema (tone, traits, behavioral constraints)
C = capsule (self-contained 5-file agent directory)
```

DAG distills knowledge from any source, augments it with typed relationships and structural classification, and generates a capsule — the atomic deployment unit. Copy the folder, copy the complete agent. The capsule is the product of DAG.

DAGR (Distillation + Augment + Generation + Retrieval) is the agent runtime pipeline, extending DAG with a verification stage. Formally:

```
DAGR(Q) = R(G(A(D(Q), KG, KB), P), KG_compiled)
```

Where:

```
Q = input query
D(Q) = Distill: extract intent, entities, format preferences
A(D, KG, KB) = Augment: retrieve relevant knowledge subgraph and KB paragraphs
G(A, P) = Generate: synthesize response via LLM with persona constraints P
R(G, KG_compiled) = Retrieve/Review: verify response against compiled KG via AEC
```

DAG creates agents. DAGR runs agents. The R in DAGR — the Retrieval stage — is where verification and self-education reside.

AEC (Agent Education Calibration) is the verification engine within the Retrieval stage. At capsule load time, AEC compiles the knowledge graph's typed nodes into executable policy checkers:

```
compile_kg: KG -> (Detectors, Blacklist, EdgePolicies)
```

Where:

```
KG = {(id, type, label, origin, edges)...} - n typed JSON-LD nodes
Detectors = {(id, patterns: set, threshold, weight)...} - one per typed node
Blacklist = {token...} - extracted from AntiPattern parenthetical terms
EdgePolicies = {(source, target, edge_type, target_blacklist)...} - compiled traversals
```

Compilation is $O(|N|)$ because each of the n nodes is visited exactly once: the label is tokenized into a set of content words (excluding stopwords), producing one detector struct per node. AntiPattern nodes additionally contribute parenthetical terms to the blacklist. Edges with verification semantics (avoids, requires, contradicts) are compiled into policy structs by visiting each edge once.

At query time, each response statement is tokenized into a set of content words. Verification is $O(|tokens_stmt|)$ per detector via set intersection: $overlap = stmt_tokens \cap detector.patterns$; $coverage = |overlap| / |detector.patterns|$. If coverage exceeds the type-specific threshold, the statement is grounded against that node. For example, statement tokens $\{css, variables, consistency\}$ intersected with detector patterns $\{css, variables, consistency, centralized\}$ yields coverage $3/4 = 0.75$, exceeding the Rule threshold of 0.50 — grounded.

When verification fails, the failure enters a self-education loop. AEC identifies the specific knowledge gaps — statements with verifiable content that matched no node. The education loop researches these specific gaps via LLM, extracts proposed triples (subject-predicate-object), and validates each through AEC at a lower threshold (0.5). Before integration, a **contradiction gate** checks each proposed triple: core nodes are indexed by subject and predicate; if a proposed acquired node matches an existing core node's subject and predicate but carries a different object value, the core node's value prevails and the proposed triple is rejected. Additionally, if a proposed triple's subject or object overlaps with AntiPattern blacklist terms by 50% or more, it is rejected — the agent cannot learn what its own rules forbid. Valid triples are integrated as acquired origin nodes, and the original query is re-evaluated against the expanded graph.

1.4 Contributions

This paper makes six contributions:

- 1 **The robot pattern as an agent architecture framework** — A structural mapping from autonomous robot competition architecture to software agent design. Every AETHER design decision traces to a specific robot subsystem: compiled navigation mesh \rightarrow knowledge graph, safety stop \rightarrow GHOST state, terrain mapping \rightarrow self-education loop, team identity \rightarrow persona, communication protocol \rightarrow definition. This pattern provides the structural foundation for all subsequent contributions.
- 1 **DAG and DAGR pipeline architecture** — Two proposed processes for agent creation (DAG: Distilled Augmented Generation) and runtime execution (DAGR: Distillation + Augment + Generation + Retrieval). DAG produces portable 5-file agent capsules through knowledge distillation from any source — SKILL.md files, research documents, markdown, or any structured text. DAGR extends DAG with a Retrieval stage for compiled verification, creating a closed loop between generation and accountability.
- 1 **Type-driven entailment via compiled knowledge graphs** — A verification architecture where the knowledge graph node's `@type` determines which verification operator fires. The `compile_kg()` function transforms the graph from a passive data store into an executable policy engine in a single $O(|N|)$ pass. Three verification layers cascade: deterministic compiled pattern matching (Layer 1), type-driven LLM operators for ambiguous cases (Layer 2), and edge policy traversal for compositional violations (Layer 3). Knowledge nodes carry provenance metadata — live URLs, access dates, validation status — enabling scheduled verification sweeps that detect stale or changed knowledge automatically.
- 1 **Autonomous self-education with integrity constraints** — When verification fails, the system identifies specific knowledge gaps, conducts targeted research, validates proposed knowledge through its own verification engine, enforces a contradiction gate where immutable core nodes hold absolute veto, and integrates validated triples. Demonstrated: AEC score improvement from 0.143 to 0.889 with 17 knowledge triples acquired, zero human intervention during the education cycle.
- 1 **Integrity mechanisms against verification gaming** — Anti-gaming (knowledge graph node identifiers stripped from LLM prompts, preventing score inflation through self-citation) and contradiction gate (core nodes hold absolute veto over acquired knowledge, preventing education loop poisoning through adversarial queries or model-pleasing behavior).
- 1 **Reproducible implementation and capsule corpus** — 15 Python files, Python standard library only, 33 capsules spanning six agent categories (factual scholars, procedural skill agents, executive advisors, validators, infrastructure, domain experts), orchestrator for automatic query routing, and a public repository with documented reproducibility commands for all experimental claims.

1.5 Paper Structure

Section 2 reviews related work across six categories and positions AETHER against existing frameworks, evaluation tools, persona standards, knowledge graph systems, self-improving agents, and constitutional AI approaches. Section 3 provides formal preliminary definitions for DAG, DAGR, AEC, the capsule model, and the knowledge graph type system. Section 4 presents the method for agent creation via the DAG process. Section 5 details the method for runtime execution and verification via the DAGR pipeline and AEC's three-layer cascade. Section 6 describes the method for autonomous learning through the self-education loop and integrity mechanisms. Section 7 covers the system architecture including orchestration, dynamic skill creation, and a brief overview of the Ψ (Psi) projection layer for UI actuation. Section 8 presents experimental results. Section 9 discusses implications, limitations, and the relationship to enterprise architectures. Section 10 concludes with broader impact and reproducibility statements.

2. Related Work

AETHER intersects six categories of existing work. Each addresses part of the agent accountability problem; none addresses the complete cycle of verified generation, autonomous education, and integrity enforcement. Table 1 summarizes the positioning.

Table 1: Related Work Taxonomy and AETHER Differentiation

Category	Representative Work	What It Solves	What It Leaves Open	AETHER's Contribution
Retrieval-Augmented Generation	Lewis et al. 2020, Guu et al. 2020	Grounding LLM output in retrieved context	No post-generation verification; retrieval quality \neq output quality	DAGR verifies <i>after</i> generation via AEC; retrieval (Augment) and verification (Retrieve) are separate stages
LLM Evaluation Frameworks	RAGAS (Es et al., 2023), G-Eval (Liu et al., 2023), TruLens, DeepEval	Scoring faithfulness, relevance, coherence	Post-hoc observation without remediation; embedding similarity fails on paraphrase; LLM-as-judge is non-deterministic	AEC compiles verification into deterministic detectors; failures feed education loop; sub-ms not seconds

Category	Representative Work	What It Solves	What It Leaves Open	AETHER's Contribution
Agent Frameworks	LangChain (Chase, 2022), CrewAI, AutoGen (Wu et al., 2023), Semantic Kernel (Microsoft, 2023)	Orchestration, tool use, multi-agent coordination	No verification layer; no learning from failure; agents are stateless across sessions	DAGR pipeline includes verification as a stage; self-education loop persists acquired knowledge in KG
Agent Safety & Constitutional AI	Constitutional AI (Bai et al., 2022), RLHF (Ouyang et al., 2022)	Constraining model behavior through training-time rules and human feedback	Rules embedded in weights, not inspectable or portable; no per-agent customization; cannot verify individual outputs against specific policies	AEC compiles inspectable, portable JSON-LD policies into per-agent verification; rules are files, not weights
Knowledge Graph Reasoning	RDF/OWL (W3C), SHACL (Knublauch & Kontokostas, 2017), TransE (Bordes et al., 2013)	Structured knowledge representation, schema validation, link prediction	SHACL validates <i>structured data</i> against shapes; KG reasoning operates on <i>graph queries</i> , not natural language	AEC validates <i>natural language</i> against typed knowledge — bridging the gap between graph-based policy and LLM output
Self-Improving Agent Systems	Voyager (Wang et al., 2023), ADAS (Hu et al., 2024), ReAct (Yao et al., 2022), AdaPlanner (Sun et al., 2023)	Skill libraries, iterative refinement, reasoning-action loops	Skills accumulate but aren't verified against structured knowledge; no contradiction detection; no immutable core	AEC validates acquired knowledge before integration; contradiction gate enforces core immutability; education is verified, not blind

2.1 Retrieval-Augmented Generation

RAG (Lewis et al., 2020) augments LLM generation with retrieved context from external knowledge stores. The standard pipeline embeds queries and documents into a shared vector space, retrieves the most similar documents, and provides them as context for generation. REALM (Guu et al., 2020) extends this with end-to-end training of the retriever.

RAG addresses the *input* side of grounding: providing relevant context to the LLM. It does not address the *output* side: verifying that the generated response actually used that context faithfully. A high-quality retrieval step does not guarantee a grounded response — the LLM may ignore, misinterpret, or selectively use the retrieved context.

AETHER's DAGR pipeline separates these concerns explicitly. The Augment stage handles retrieval (input grounding). The Retrieve/Review stage handles verification (output checking). They are distinct pipeline stages with distinct functions, connected but not conflated.

2.2 LLM Evaluation Frameworks

RAGAS (Es et al., 2023) is the most widely adopted RAG evaluation framework, computing faithfulness, answer relevance, and context precision using embedding similarity and LLM-based scoring. G-Eval (Liu et al., 2023) uses GPT-4 chain-of-thought evaluation with human-aligned rubrics. TruLens provides guardrail-based evaluation with customizable feedback functions. DeepEval offers an open-source alternative with similar metrics.

These tools share three structural limitations for agent verification:

First, **embedding similarity fails on paraphrased content**. LLMs routinely restructure and rephrase retrieved content — this is their normal generation behavior. In testing, cosine similarity between "72 degrees Fahrenheit" and the reference "72[°]F" falls well below 50% despite identical semantic content. Between "approximately fifteen million dollars" and "\$15,000,000," similarity is low despite identical meaning. Any verification system built on text similarity will systematically undercount grounding when the LLM paraphrases — which is always.

Second, **LLM-as-judge introduces non-determinism**. The same response evaluated twice by the same judge model may receive different scores. When the judge is the same model that generated the response — common in production due to cost constraints — structural generosity bias means the system tends to approve its own work. We observed this directly when using Claude to verify Claude's output. This concern applies to any system using an LLM in its verification path — including AETHER's Layer 2, which invokes an LLM for ambiguous cases. AETHER's primary defense is that Layer 1 (compiled pattern matching) and Layer 3 (edge policy traversal) are fully deterministic and resolve the majority of statements without any LLM involvement. Layer 2 is a constrained fallback with a generosity guard, not the primary verification mechanism. This limitation is addressed further in Section 9.6.

Third, **evaluation without remediation is incomplete**. A RAGAS faithfulness score of 0.4 tells you the response is poorly grounded. It does not tell you *which specific knowledge* is missing, does not research that knowledge, and does not prevent the same failure on the next query. The evaluation is diagnostic but not therapeutic.

AEC addresses all three: verification via compiled set intersection (not embedding similarity), deterministic scoring for the primary verification path (Layers 1 and 3), and failure-driven education (not just scoring but remediation through the self-education loop).

2.3 Agent Frameworks

LangChain (Chase, 2022) provides chains, agents, and tools as composable abstractions for LLM applications. CrewAI (Joao et al., 2024) organizes agents into role-based crews with task

delegation. AutoGen (Wu et al., 2023) enables multi-agent conversations with human-in-the-loop patterns. Semantic Kernel (Microsoft, 2023) provides an SDK for integrating LLMs into applications with plugin architectures.

These frameworks excel at orchestration — defining what agents do, how they communicate, and how they use tools. But their use of the word "skill" obscures a fundamental structural limitation. A LangChain agent is a function router. A CrewAI agent is a prompt template with memory bolted on. A Semantic Kernel "skill" is literally a C# method. None have knowledge that belongs to the agent. None persist identity when the process stops. None improve from failure. As orchestration tools, they are well-suited to their purpose; as agents-with-skills, they are hammers pretending to be carpenters.

AETHER's capsule model is fundamentally different. The knowledge graph persists across sessions. The education loop writes acquired knowledge back to the graph. The compiled verification engine is rebuilt at each load to incorporate new knowledge. The agent that ran 100 queries is measurably different from the agent that ran 10, because its knowledge graph has grown through verified self-education. Point the same capsule at Claude, GPT-4, or Gemini — the knowledge, verification, and identity travel intact. The model is the engine. The skill is the asset.

2.4 Agent Safety and Constitutional AI

Constitutional AI (Bai et al., 2022) constrains model behavior through a set of principles applied during training. The model learns to self-critique and revise its outputs according to a constitution. RLHF (Ouyang et al., 2022) aligns model behavior with human preferences through reinforcement learning from human feedback.

These approaches operate at the *model* level — the constraints are embedded in the model's weights through training. This has three implications that AEC addresses differently:

First, constitutional constraints are **not inspectable** at inference time. You cannot examine a trained model and determine exactly which rules it follows. AEC's knowledge graph is a JSON-LD file — human-readable, machine-parseable, auditable.

Second, constitutional constraints are **not portable** between models. Training Claude with a constitution does not transfer those constraints to GPT-4. AEC's verification is model-independent — the same capsule with the same KG produces the same verification results regardless of which LLM generated the response.

Third, constitutional constraints are **not per-agent customizable**. Every instance of Claude follows the same constitution. AEC's verification rules are per-capsule — a medical agent and a legal agent can have entirely different verification policies encoded in their respective knowledge graphs.

2.5 Knowledge Graph Reasoning

RDF (W3C, 2014) and OWL (W3C, 2012) provide formal ontology languages for structured knowledge representation. SHACL (Knublauch & Kontokostas, 2017) validates RDF graph data against structural constraints — shapes that define which properties a node must have, cardinality restrictions, and value type constraints. Knowledge graph embedding methods like TransE (Bordes et al., 2013) and RotatE (Sun et al., 2019) learn vector representations for link prediction and graph completion.

SHACL is the closest prior art to AEC's compilation approach. Both share the principle that the *schema drives the validation logic* — verification rules are derived from structural definitions, not learned or manually coded. However, SHACL validates **structured data** against shapes: does this RDF node have the required properties with valid values? AEC validates **natural language** against typed knowledge: does this LLM-generated sentence comply with the rules encoded in these typed graph nodes?

This is the specific gap AEC bridges. The input to SHACL is a graph node. The input to AEC is a sentence. The verification logic for SHACL is shape conformance. The verification logic for AEC is type-driven entailment — the node's `@type` determines which natural language verification operator fires. The output of both is a compliance determination, but they operate on fundamentally different inputs.

FactKG (Kim et al., 2023) addresses fact verification via reasoning on knowledge graphs — verifying claims against structured KG evidence using multi-hop paths and logical operators. This is the closest prior work to AEC's verification model. The distinction is input format: FactKG verifies structured claims against a shared knowledge graph (Wikidata, DBpedia); AEC verifies free-form LLM-generated natural language against the agent's own typed, compiled knowledge graph. FactKG operates at the dataset level; AEC operates per-agent, per-response, in real-time at sub-millisecond latency.

One acknowledged limitation of the robot pattern analogy: autonomous robots have physical sensors that ground them in the real world — encoders, cameras, lidar. AETHER's "sensors" are API calls and file reads — disembodied grounding through text. The pattern holds for cognitive architecture but does not solve the symbol grounding problem. AEC verifies consistency with stated knowledge, not correspondence with physical reality. This limitation is noted explicitly in Section 9.

AETHER uses JSON-LD (W3C, 2014) as its knowledge graph format specifically because it provides the structured typing that AEC requires for compilation while remaining compatible with the broader linked data ecosystem. The JSON-LD format enables native interoperability with enterprise A2A communication protocols without format translation.

2.6 Self-Improving Agent Systems

Voyager (Wang et al., 2023) maintains a skill library that grows through exploration in Minecraft — discovered behaviors are stored as executable code for reuse. ADAS (Hu et al., 2024) proposes meta-agents that iteratively program, evaluate, and refine new agents. ReAct (Yao et al., 2022)

implements reasoning-action loops where agents update their context through observation. AdaPlanner (Sun et al., 2023) refines plans through feedback from environmental interaction.

These systems share a common pattern: improvement through operational experience. They differ from AETHER in three respects.

First, **accumulated skills are not verified against structured knowledge**. Voyager adds code to its skill library when it works in practice. It does not check whether the code is consistent with any formal specification. AEC validates proposed knowledge against the compiled policy engine before integration — the education loop is verified, not blind.

Second, **there is no contradiction detection**. When a self-improving agent acquires knowledge that conflicts with its existing knowledge, most systems either overwrite silently or accumulate contradictions. AETHER's contradiction gate explicitly checks proposed acquired nodes against immutable core nodes: same subject and predicate with different object values triggers a rejection. The agent cannot unlearn its foundational knowledge through operational experience.

Third, **there is no immutable core**. In Voyager, all skills are equally mutable. In OpenClaw, the agent can edit its own SOUL.md. AETHER distinguishes between core knowledge (immutable at runtime, representing the agent's foundational training) and acquired knowledge (mutable, representing operational learning). The core/acquired distinction is enforced by the contradiction gate and is structurally visible in the JSON-LD origin types.

2.7 Concurrent Work

Bi et al. (2026) propose a framework for automated acquisition of agent skills through mining open-source repositories, translating extracted procedural knowledge into standardized SKILL.md format. Their work addresses skill *creation* — how to produce agent skills from existing codebases — but not skill *verification* or *improvement*. A skill extracted from a GitHub repository has no mechanism to verify that an agent using that skill follows its procedures, and no mechanism to improve the skill through operational feedback.

AETHER's DAG process addresses the same creation challenge (distilling knowledge into agent artifacts) while DAGR adds the verification and education dimensions that Bi et al. do not address. The two approaches are complementary: repository mining could serve as an additional input source for the DAG process, with AEC providing the verification layer that extracted skills currently lack.

3. Preliminaries

This section formalizes the core definitions that govern the remainder of the paper. We begin with the conceptual foundation — what distinguishes a skill from a tool — then define the capsule as the structural realization of that distinction, and establish the formal notation for DAG, DAGR, AEC, and the knowledge graph type system.

3.1 What Is a Skill?

A skill is something an entity **knows**, **expresses consistently**, and **gets better at through practice**.

This definition is deliberately distinct from what agent frameworks currently ship. A function call is not a skill — it has no knowledge, no consistency of expression, no improvement through use. A prompt template is not a skill — it has knowledge but no identity, no persistence, no self-correction. A fine-tuned model has absorbed skills into its weights, but those skills are not inspectable, not portable, and not independently improvable.

A skill, properly defined, requires five properties:

Property	Meaning	Agent Equivalent
Knowledge	Domain-specific information the skill draws from	Structured + unstructured knowledge base
Identity	Persistent, versioned existence independent of runtime	Manifest with UID and lineage
Expression	Consistent voice, reasoning style, behavioral traits	Persona definition
Behavior	Configurable operational parameters and boundaries	Pipeline configuration and domain limits
Improvement	Gets better from practice; failures inform growth	Verified self-education loop

No mainstream agent framework satisfies all five. Most satisfy zero or one.

3.2 The Capsule: A Skill as an Agent

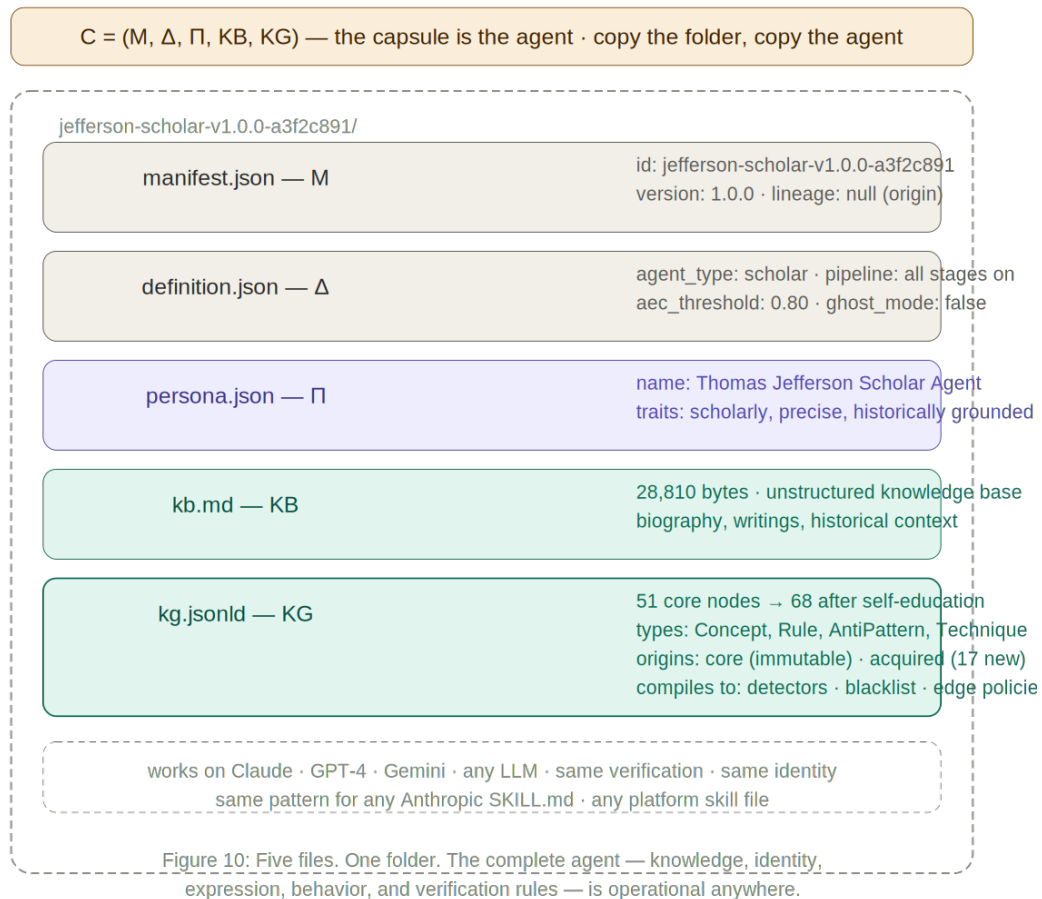
The AETHER capsule is the structural realization of all five skill properties as files in a directory. Each file maps directly to an autonomous robot subsystem — the architectural pattern described in Section 1.1:

Skill Property	Capsule File	Robot Analog	Format	Mutability
Knowledge (structured)	<code>{id}-kg.jsonld</code>	Compiled navigation mesh	Typed JSON-LD graph	Core: immutable. Acquired: mutable via education.
Knowledge (unstructured)	<code>{id}-kb.md</code>	Operator field notes	Markdown + provenance URLs	Immutable core
Identity	<code>{id}-manifest.json</code>	Serial number + lineage	JSON	Updated on restamp only

Skill Property	Capsule File	Robot Analog	Format	Mutability
Expression	{id}-persona.json	Team identity and role	JSON	Configurable
Behavior	{id}-definition.json	Communication protocol	JSON	Configurable
Improvement	Education loop + AEC	Terrain mapping mechanism	Runtime mechanism	Writes acquired nodes to KG

The capsule is not an agent that has skills. The capsule is a skill that is an agent.

The folder exists. The agent exists. They are the same thing. No runtime, no framework, no process required for the agent to *be* — only for it to *act*. Copy the folder to another machine, point it at any LLM, and the complete agent — knowledge, identity, expression, behavior, and verification rules — is operational.



3.3 Formal Definitions

Definition 1 (Capsule). A capsule C is a 5-tuple:

```
C = (M, Delta, Pi, KB, KG)
```

Where:

```
M = manifest (id, name, version, created, lineage)
Delta = definition (pipeline config, thresholds, domain boundaries, triggers)
Pi = persona (tone, style, traits, description)
KB = knowledge base (unstructured markdown)
KG = knowledge graph (typed JSON-LD)
```

Definition 2 (Knowledge Graph). A knowledge graph KG is a set of typed nodes with edges:

```
KG = (N, E, T, O)
```

Where:

```
N = {n_1, n_2, ..., n_k} - nodes
E = {(n_i, relation, n_j)...} - directed typed edges
T: N -> {Concept, Rule, AntiPattern, Technique, Tool, Trait} - type function
O: N -> {core, acquired, updated, deprecated, provenance} - origin function
```

The type function T is not metadata. It is a verification instruction — the mechanism by which the knowledge graph compiles into executable logic (Section 5).

The origin function O enforces the core/acquired distinction and tracks the provenance of every node:

Origin	How it enters the KG	Mutable	Human equivalent
core	Created during DAG distillation from source material	Never	What you learned in formal training
acquired	Added by the self-education loop after AEC failure	Prunable	"I didn't know this, researched it, now I do"
updated	Refreshed by scheduled sweep against live source	Replaces stale	"I knew this, confirmed it's still true"
deprecated	Flagged when source is dead or content has changed	Frozen, not deleted	"I knew this as fact but it changed"
provenance	Citation metadata: live URL, access date, validation status	Updated by sweeps	"I know this because I read it here, on this date"

Provenance nodes carry live URLs. Scheduled maintenance sweeps — designed but not yet fully automated — validate whether those URLs are still live and whether the source content has changed. A dead URL flags all connected claim nodes. Changed content is designed to trigger the self-education loop automatically; the flagging mechanism is implemented, with the auto-trigger connection representing near-term future work (Section 10.2). This provenance architecture is the mechanism that keeps the knowledge graph current — the agent's knowledge has a biography, not just a value.

Nodes with $O(n) = \text{core}$ are immutable at runtime. Nodes with $O(n) = \text{acquired}$ are mutable and subject to the contradiction gate — an algorithm that checks proposed acquired nodes against core nodes before integration (defined in Section 6). This structural separation enables self-education without knowledge corruption.

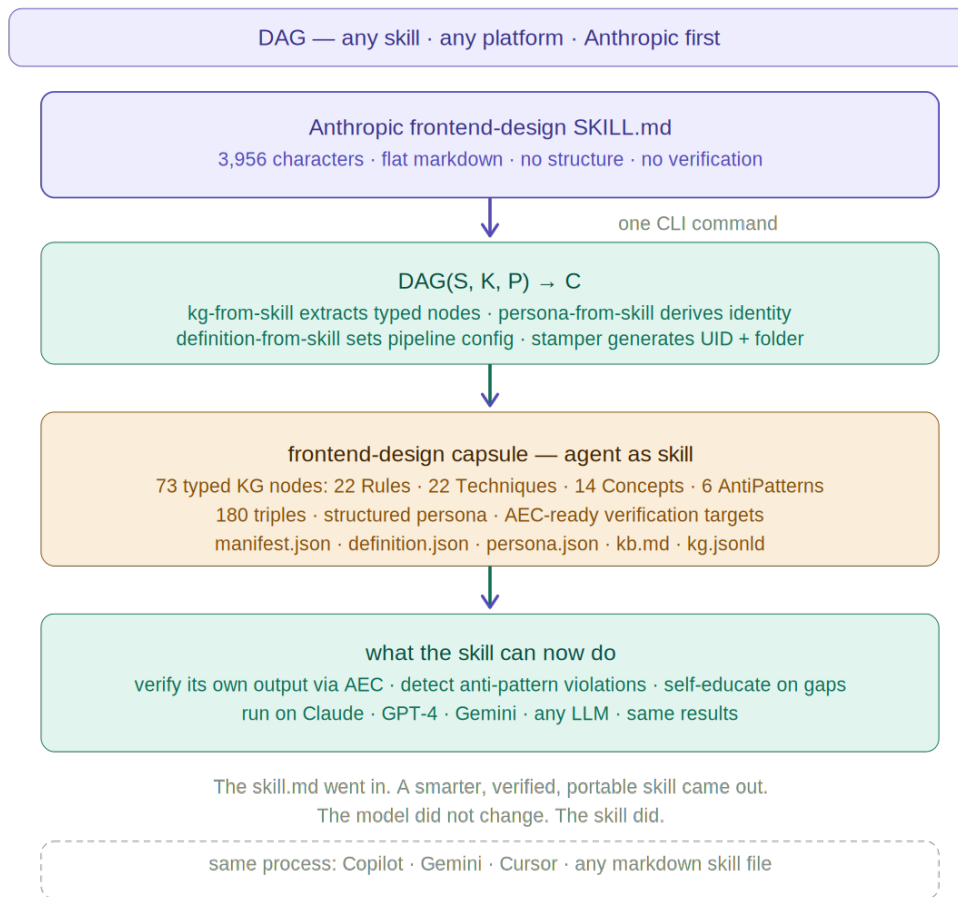
Definition 3 (DAG — Distilled Augmented Generation). The agent creation process.

Running example: Consider a frontend-design SKILL.md containing rules like "Avoid generic fonts (Inter, Roboto, Arial)" and techniques like "Use staggered animation reveals." DAG distills this into a capsule with 73 typed KG nodes — 22 Rules, 22 Techniques, 14 Concepts, 6 AntiPatterns — each carrying verification semantics that AEC will compile into policy checkers. This capsule and its verification behavior will illustrate each definition that follows.

```
DAG: (S, K, P) -> C

Where:
  S = source material in {SKILL.md, research document, markdown, repository}
  K: S -> (KB, KG) - knowledge distillation function
    K decomposes into:
      K_text: S -> KB - extract unstructured knowledge as markdown
      K_graph: S -> KG - extract typed nodes and edges as JSON-LD
  P: S -> Pi - persona extraction function
  C = (M_generated, Delta_extracted, P(S), K_text(S), K_graph(S))
```

DAG distills knowledge from any source, augments it with typed relationships and structural classification, and generates a capsule. The manifest M is generated by the stamper with a unique identifier. The definition Δ is extracted from the source material's domain boundaries and behavioral indicators.



Definition 4 (DAGR — Distillation + Augment + Generation + Retrieval). The agent runtime pipeline:

```

DAGR: Q x C -> (response, score, gaps, KG')

Where:
Q = input query
C = capsule (M, Delta, Pi, KB, KG)

D(Q) -> (intent, entities, format)          - Distill
A(D(Q), KG, KB) -> context                  - Augment
G(context, Pi) -> response                   - Generate (LLM call)
R(response, KG_compiled) -> (score, gaps)    - Retrieve/Review (AEC)

DAGR(Q, C) = R(G(A(D(Q), KG, KB), Pi), compile(KG))

If score >= threshold: return (response, score, {}, KG)
If score < threshold: return (response, score, gaps, KG)
                    and queue (Q, response, gaps) for education
  
```

DAG creates the capsule. DAGR runs the capsule. The R in DAGR — the Retrieval stage — is where verification and the education trigger reside.

Definition 5 (AEC — Agent Education Calibration). The verification engine:

```

compile: KG -> (D_set, B, P_edge)

Where:
  D_set = {(n_i, patterns_i, threshold_i, weight_i)...} - compiled detectors
    patterns_i = tokenize(label(n_i)) \ stopwords
    threshold_i = f(T(n_i)) - type-specific threshold
    weight_i = g(T(n_i)) - type-specific weight

  B = U{parenthetical_terms(label(n)) : T(n) = AntiPattern} - blacklist

  P_edge = {(n_i, n_j, relation, B_target)...} - edge policies
    for each (n_i, relation, n_j) where relation in {avoids, requires, contradicts}

Compile time: O(|N|) - each node visited once

```

```

verify: (response, D_set, B, P_edge) -> {(statement, category, evidence)...}

Where:
  category in {grounded, ungrounded, persona}

For each statement s in split(response):
  tokens_s = tokenize(s) \ stopwords

  Layer 1 (deterministic):
    If tokens_s n B != {} -> (s, ungrounded, blacklist_hit)
    For each d in D_set:
      coverage = |tokens_s n d.patterns| / |d.patterns|
      If coverage >= d.threshold -> (s, grounded, d.node_id)

  Layer 2 (type-driven, if Layer 1 inconclusive and LLM available):
    Select top-3 candidates from D_set by partial coverage
    For each candidate:
      classification = LLM_classify(s, candidate, T(candidate))
      - prompts the LLM: "Does statement s FOLLOW, VIOLATE, or have
        NO RELATION to this [T(candidate)] node?" Forces structured
        JSON output. Includes generosity guard: "If merely good advice
        but not explicitly supported, classify UNRELATED."
      If classification = FOLLOW -> (s, grounded, candidate.node_id)
      If classification = VIOLATE -> (s, ungrounded, candidate.node_id)

  Layer 3 (edge traversal, if Layer 1/2 matched a node):
    For each matched node n_i:
      For each (n_i, avoids, n_j) in P_edge:
        If tokens_s n B_target(n_j) != {} -> (s, ungrounded, edge_path)

    Default: (s, persona, no_match)

Runtime per statement: O(|tokens_s|) for set intersection operations

```

```

score: results -> [0, 1]

score = |{r : r.category = grounded}| / |{r : r.category in {grounded, ungrounded}}|

Persona statements excluded from both numerator and denominator.
If all statements are persona: score = 1.0 (all-persona pass)

```

3.4 Knowledge Graph Type System

The type function T maps each node to exactly one of six types. Each type carries verification semantics — it determines how AEC evaluates statements that match the node:

Type	Verification Semantics	Threshold	Weight	AEC Behavior
Concept	Relevance detection	0.30	0.5	Does the statement address this domain?
Rule	Compliance checking	0.50	1.0	Does the statement follow this guideline?
AntiPattern	Violation detection	0.40	1.0	Does the statement use what this pattern forbids?
Technique	Application verification	0.55	1.0	Is this technique being applied?
Tool	Usage context	0.60	0.3	Is this tool referenced appropriately?
Trait	Persona alignment	0.70	0.2	Is the expression consistent with this trait?

These thresholds were tuned empirically against the 73-node frontend-design skill agent using a 50-query validation set spanning rule compliance, anti-pattern violation, and technique application scenarios. Concepts receive the lowest threshold (0.30) because domain relevance can be expressed with few overlapping terms. Rules and AntiPatterns receive the highest weight (1.0) because policy compliance is the primary verification target. The threshold values represent the minimum token coverage ratio required for a Layer 1 match to be classified as grounded. Broader validation across diverse knowledge graphs is ongoing (Section 9).

3.5 Edge Types and Verification Semantics

Three edge types carry verification semantics — they compile into Layer 3 policy functions:

Edge	Formal	Verification Logic
avoids	$(n_i, \text{avoids}, n_j)$	If statement matches n_i AND tokens overlap with n_j blacklist \rightarrow violation
requires	$(n_i, \text{requires}, n_j)$	Informational in v1 — logged if n_j absent, not scored
contradicts	$(n_i, \text{contradicts}, n_j)$	If statement matches both n_i AND n_j \rightarrow contradiction

Three additional edge types provide structural organization but do not compile into verification policies:

Edge	Purpose
enables	n\$_i\$ makes n\$_j\$ possible (routing signal)
contains	Hierarchical grouping
prioritizes	Relative importance (routing tiebreaker)

The distinction between verification edges and structural edges is architecturally significant: `compile_kg()` processes only the first three, keeping compilation time strictly proportional to the number of verification-relevant edges, not the total edge count.

4. Method: Agent Creation via DAG

This section describes how agents are created through the DAG (Distilled Augmented Generation) process. DAG transforms source material — a SKILL.md file, a research document, a markdown reference, or any structured text — into a complete capsule. The process is automatic: one CLI command, one source file, one output directory.

4.1 The Ingest Pipeline

DAG is implemented through three ingest modes, each optimized for a different source format:

Mode	Source	LLM Required	Use Case
<code>ingest_skill</code>	SKILL.md with YAML frontmatter	Yes	Claude/Copilot skill files, SoulSpec agents
<code>ingest_research</code>	Structured deep research output	No	Gemini Deep Research, structured reports
<code>ingest_document</code>	Any markdown file	Yes	Documentation, guidelines, references

All three modes produce the same output: a valid 5-file capsule directory. The difference is in how knowledge is extracted — `ingest_research` parses deterministically (no LLM call), while `ingest_skill` and `ingest_document` use LLM-assisted extraction.

Returning to the running example: The frontend-design capsule was created via `ingest_skill` from Anthropic's official frontend-design SKILL.md (3,956 characters, published as part of Claude's 17 official skills). A single CLI command:

```
python cli.py ingest-skill input/skills/frontend-design/SKILL.md \
  -output ./examples -provider anthropic
```

This produced a complete capsule in `examples/frontend-design-v1.0.0-ff6ab491/` with 73 KG nodes, 180 triples, and a structured persona — from a flat markdown file to a

self-verifying agent in one invocation.

4.2 Knowledge Distillation: Three Extraction Skills

The critical step in DAG is knowledge distillation: transforming unstructured source text into a typed knowledge graph. Generic LLM prompts produce thin, poorly typed graphs. AETHER addresses this with **extraction skills** — specialized prompt templates that themselves follow the SKILL.md format.

Three extraction skills govern the distillation process:

kg-from-skill extracts the knowledge graph. It instructs the LLM to identify six node types (Concept, Rule, AntiPattern, Technique, Tool, Trait), establish relationship edges between them (avoids, requires, enables, contradicts, contains, prioritizes), and produce valid JSON-LD with the AETHER @context. The prompt specifies that every "never," "always," "must," and "avoid" in the source should become Rule or AntiPattern nodes — capturing policy knowledge that will later compile into AEC verification detectors.

persona-from-skill derives the agent's personality. It reads the source material's language — bold assertions suggest confidence, warning-heavy text suggests caution, creative direction suggests artistry — and produces a structured persona with named traits, keywords, and a distinctive identity.

definition-from-skill extracts behavioral configuration. Domain boundaries (what the agent is authoritative on, what it must defer), trigger conditions (when should this agent activate), and suggested AEC verification gates specific to this domain.

The extraction skills are themselves SKILL.md files — AETHER using its own skill format to build its own agents. This meta-circularity is a concrete demonstration of the skill-as-agent thesis: the extraction skills are agents that create agents.

4.3 Extraction Quality

The specialized extraction skills produce significantly richer knowledge graphs than generic LLM prompts. On the same frontend-design source material:

Extraction Method	Total Nodes	Rules	AntiPatterns	Techniques	Concepts	Edges
Generic prompt	50	14	3	11	18	~60
Specialized kg-from-skill	73	22	6	22	14	180
Improvement	+46%	+57%	+100%	+100%	-22%	+200%

The specialized skill produces 46% more nodes overall, with the most significant gains in the categories that matter most for AEC verification: Rules (+57%), AntiPatterns (+100%), and Techniques (+100%). The decrease in Concept nodes reflects better classification — nodes that generic prompts loosely typed as Concepts were correctly identified as specific Rules or Techniques by the specialized prompt.

Edge density increased by 200%, which directly impacts Layer 3 verification: more edges mean more compiled policy functions, which means richer compositional violation detection.

4.4 The Stamper

Once knowledge distillation produces the five capsule files, the **stamper** generates the capsule directory:

- 1 Computes a unique identifier from name + version + timestamp:
`{slug}-v{version}-{uid8}`
- 2 Creates the directory in the output path
- 3 Names all files with the capsule ID prefix (ensuring files are identifiable outside their directory)
- 4 Records lineage: `previous_id` and `previous_version` if this is a restamp of an existing capsule
- 5 Validates the capsule (all five files present and parseable)

The stamper is the "birth certificate" mechanism. Every capsule has a provenance chain — who was its parent, what version did it descend from. When capsules improve through education and are re-stamped as new versions, the lineage record creates an auditable evolution trail.

4.5 Skill Portability: Any Source, Same Output

DAG's three ingest modes share a common output format: the 5-file capsule. This means any source of agent knowledge — regardless of its origin format — produces the same portable artifact.

Source Ecosystem	Input Format	DAG Mode	Output
Claude Code	SKILL.md	<code>ingest_skill</code>	Capsule
Copilot	SKILL.md / .agent.md	<code>ingest_skill</code>	Capsule
SoulSpec / OpenClaw	SOUL.md + soul.json	<code>ingest_skill</code> (planned)	Capsule
Gemini Deep Research	Structured markdown	<code>ingest_research</code>	Capsule
Enterprise documentation	Markdown / PDF	<code>ingest_document</code>	Capsule
GitHub repositories	Code + README	Planned	Capsule

The source format is consumed. The capsule is produced. The skill has been distilled, augmented with typed structure, and generated as an agent. DAG complete.

All 17 of Anthropic's official Claude Code skills were ingested through DAG, producing 17 complete capsules — each with typed knowledge graphs, structured personas, and AEC-ready verification targets. The frontend-design skill — 3,956 characters of flat markdown — became a 73-node typed knowledge graph with rule compliance checking, anti-pattern violation detection, and self-education capability in a single CLI invocation, and serves as the fully validated representative example. The remaining 16 capsules were produced through the same DAG process and are included in the repository; detailed per-capsule validation follows the same pattern with frontend-design as the reference implementation. The skill.md went in. A smarter, verified, portable skill came out. An additional 8 executive advisory capsules were created through a collaborative DAG process using Gemini for knowledge research and the AETHER extraction skills for graph generation.

The platform ingestion matrix is designed for bidirectional operation. A future export layer will allow capsules to publish their improved knowledge back to the originating platform — an AETHER-improved capsule exporting an updated SKILL.md back to Claude Code, or an improved agent.md back to Copilot. The skill enters AETHER, improves through verified self-education, and returns to its platform smarter than it arrived.

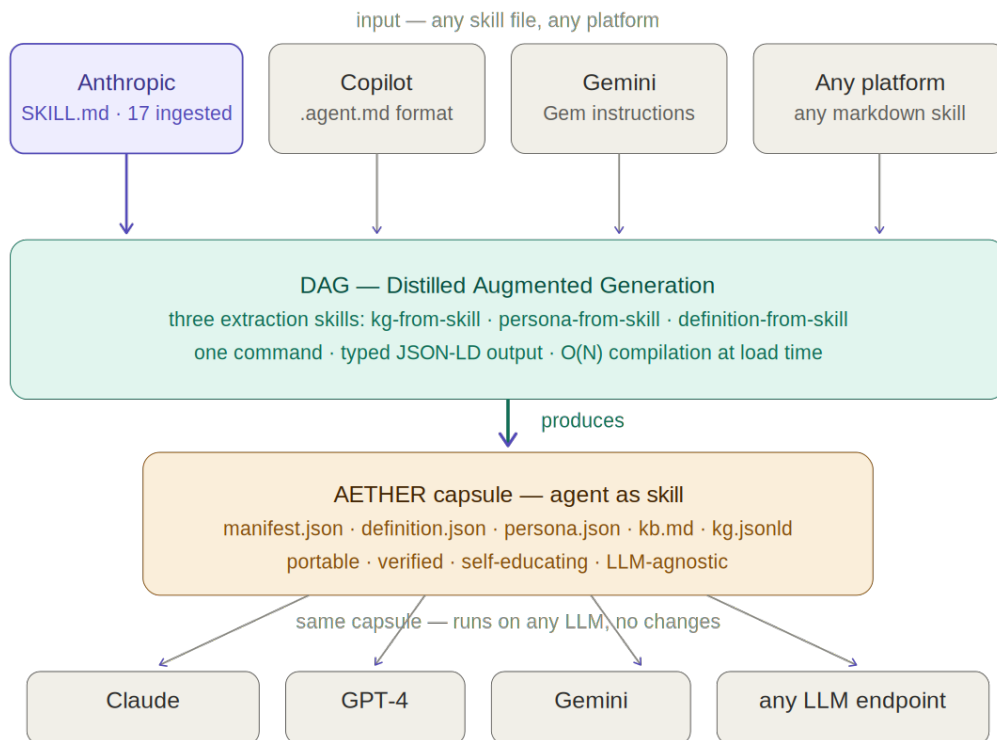


Figure 2: The skill.md went in. A smarter, verified, portable skill came out.
The model is the motor. The skill is the asset.

4.6 What DAG Does Not Do

DAG creates the capsule. It does not run the capsule (that is DAGR, Section 5). It does not verify the capsule's output (that is AEC within DAGR). It does not improve the capsule (that is the self-education loop, Section 6).

A capsule produced by DAG is operational but unrefined. Its knowledge graph reflects the source material faithfully, but the thresholds may need tuning, the coverage may have gaps, and the anti-patterns may be incomplete. This is by design — the DAGR pipeline and self-education loop handle refinement through use. DAG produces the seed. DAGR and AEC grow it.

This separation is architecturally significant: it means DAG can be fast and automated (one command, seconds to complete) without requiring the capsule to be perfect at creation. The education loop closes the quality gap over time. The first response from a DAG-created capsule will be verified by AEC. If it fails, education begins. The skill improves through practice — exactly as the skill definition in Section 3.1 requires.

5. Method: Runtime Execution and Verification via DAGR

This section describes how agents operate at runtime through the DAGR (Distillation + Augment + Generation + Retrieval) pipeline. The focus is the Retrieval stage — where AEC (Agent Education Calibration) compiles the knowledge graph into executable verification logic and evaluates every generated response.

5.1 The DAGR Pipeline

A query enters the capsule and passes through four stages. Each stage receives a shared context dictionary, adds its results, and passes forward.

Stage 1: Distill. Extracts structure from the raw query without an LLM call. Using regex and keyword patterns, Distill identifies: intent (question, instruction, comparison, creation, or general), named entities (capitalized terms), format preference (list, table, JSON, prose), and brevity flag. Processing time: <0.1ms. The output narrows the search space for the Augment stage.

*Running example: Query "How should I approach typography for a luxury brand?"
\$ \rightarrow \$ intent: general, entities: {typography, luxury, brand}, format: prose, brevity: false.*

Stage 2: Augment. Retrieves relevant knowledge from the capsule's KB and KG. KB paragraphs are scored by entity overlap against the distilled entities (top-3 returned). KG nodes are matched by property value overlap. The matched content becomes the grounding context for generation.

Running example: Augment retrieves 3 KB paragraphs discussing typography principles and 5 KG nodes including concept:typography, concept:aesthetic_vision, trait:bold_creative.

Anti-gaming note: KG nodes included in the prompt have their `@id` values and all `aether:` namespaced properties stripped. The LLM receives human-readable labels ("Avoid generic fonts like Arial and Inter") but not node identifiers (`rule:avoid_generic_fonts`). This prevents the LLM from embedding verification labels in its response to inflate AEC scores. AEC in the Retrieval stage retains full access to all fields. This mechanism is detailed in Section 6.

Stage 3: Generate. Constructs a prompt from three components: persona instructions (from `persona.json` — tone, style, behavioral constraints), matched KB and KG context (from Augment, with node IDs stripped), and the original query. A single LLM API call produces the response. This is the only stage that makes an external call, the only stage that costs money, and the only stage that introduces non-determinism. All other stages are pure Python stdlib operations.

Running example: The LLM receives the persona ("bold creative, visionary technical"), the matched typography context, and the query. It generates a response recommending Playfair Display, Crimson Pro, and Space Grotesk with CSS implementation details and a negative letter-spacing technique.

Stage 4: Retrieve/Review. AEC verifies the generated response against the compiled knowledge graph. This is the R in DAGR — the stage that closes the accountability loop. If the response passes the threshold (default 0.80), it is delivered. If it fails, the response is delivered with a below-threshold warning, and the failure enters the education queue (Section 6).

Running example: AEC evaluates 8 statements. 6 are grounded against KG Rules and Techniques via compiled pattern matching. 1 triggers an anti-pattern violation ("Inter" detected in blacklist). 1 is classified as persona (genuine opinion). Score: 0.857. Threshold: 0.80. PASS.

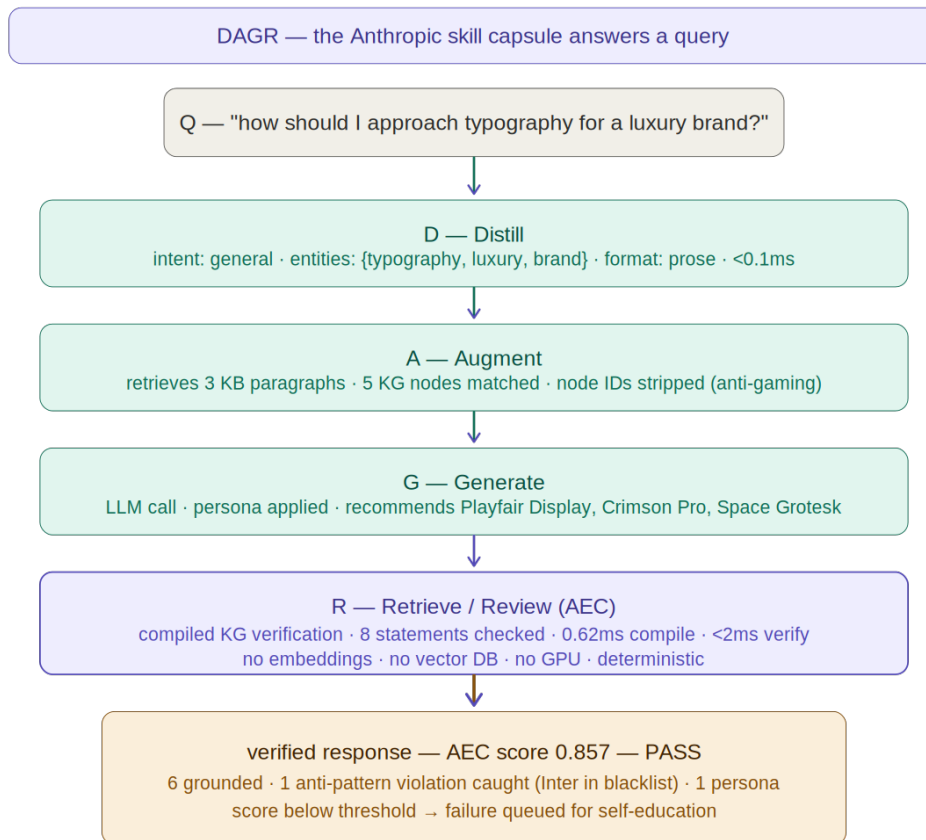


Figure 4: $\text{DAGR}(Q) = R(G(A(D(Q), \text{KG}, \text{KB}), P), \text{compile}(\text{KG}))$
Every response from every Anthropic skill capsule passes through this pipeline.

5.2 AEC Compilation: The Knowledge Graph as Program

The central mechanism of AEC is `compile_kg()` — a function that transforms the knowledge graph from a data structure into an executable verification program. This function runs once when the capsule loads. All subsequent verification uses the compiled output.

What compilation produces:

```

compile_kg(KG) -> {
  detectors:    [{node_id, label, node_type, patterns: set, threshold, weight}...],
  blacklist:    {token...},
  blacklist_map: {token -> {node_id, label}...},
  edge_policies: [{source_id, source_patterns, target_id, target_patterns,
                    target_blacklist, edge_type}...],
  node_lookup:  {node_id -> node...}
}

```

How compilation works:

For each of the n nodes in the knowledge graph:

- 1 The `rdfs:label` is tokenized into a set of content words, excluding stopwords. This produces the `patterns` set — the compiled representation of what this node "means" at the token level.
- 2 The node's `@type` determines the `threshold` and `weight` via the type configuration table (Section 3.4).
- 3 If the node is an `AntiPattern`, parenthetical terms from the label are extracted into the global blacklist. "Overused fonts (Inter, Roboto, Arial)" contributes `{inter, roboto, arial}` to the blacklist. Only parenthetical terms are extracted — this prevents common words in `AntiPattern` descriptions from triggering false violations.
- 4 If the node has outgoing `skill:avoids`, `skill:requires`, or `skill:contradicts` edges, those edges compile into edge policy structs containing source patterns, target patterns, and target blacklists.

Compilation complexity: $O(|N| + |E_v|)$ where N is the number of nodes and E_v is the number of verification-relevant edges. Each node is visited exactly once for tokenization. Each verification edge is visited once for policy compilation. On the 73-node frontend-design knowledge graph, compilation completes in 0.62ms, producing 73 detectors, 5 blacklist tokens, and 12 edge policies.

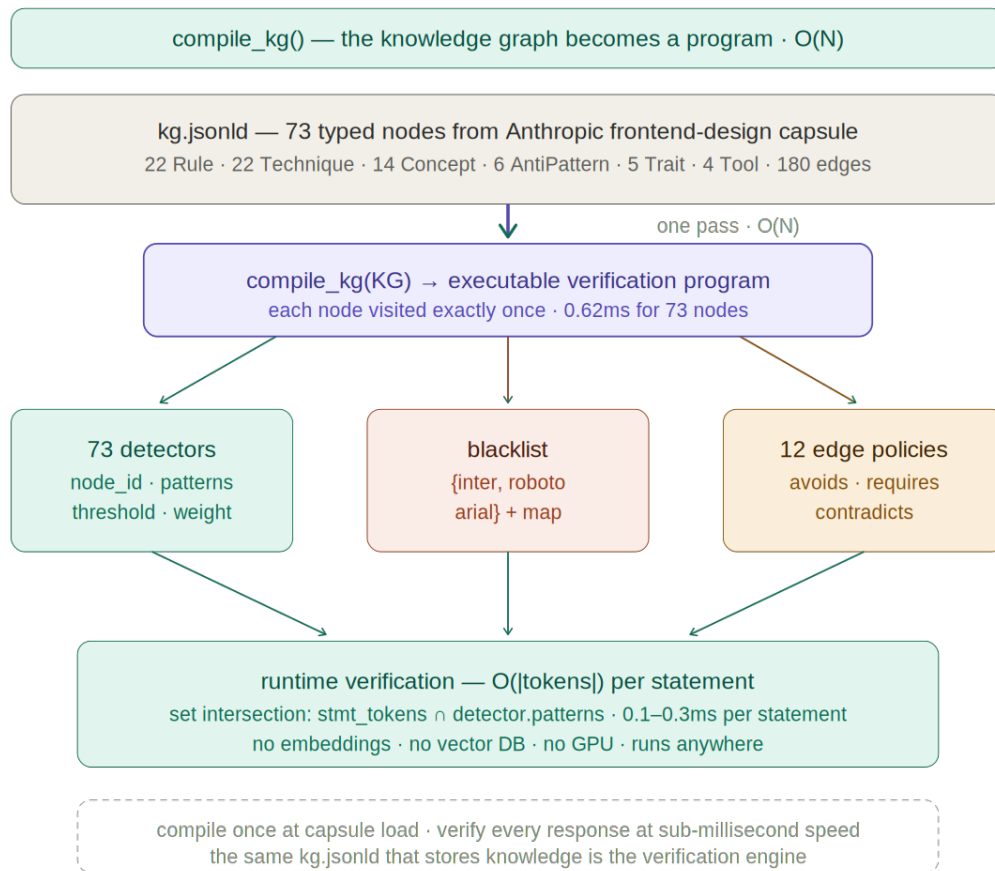


Figure 6: The knowledge graph is not a lookup table. It is a compiled program.

5.3 Layer 1: Compiled Pattern Matching

Layer 1 is entirely deterministic. No LLM call. It resolves the majority of statements.

For each statement in the response:

- 1 **Tokenize** the statement into content words, excluding the same stopword set used during compilation.
- 2 **Blacklist check.** Compute `statement_tokens & blacklist`. If non-empty, the statement is immediately classified as **ungrounded** with the matching `AntiPattern` as evidence. This is the fastest check — $O(1)$ per token via set membership.
- 3 **Detector scan.** For each compiled detector, compute $\text{coverage} = |\text{statement_tokens} \cap \text{detector.patterns}| / |\text{detector.patterns}|$. If coverage exceeds the detector's type-specific threshold, the statement is **grounded** against that node. The best match (highest coverage) is selected.
- 4 **Ambiguous range.** If no detector exceeds its threshold but at least one exceeds half the threshold, the statement and its top candidates are tagged for Layer 2 evaluation. Additionally, Sørensen-Dice coefficient (bigram similarity) is computed as a secondary signal for candidates in this range.

Running example: Statement "Use CSS variables for consistency" \rightarrow tokens: {css, variables, consistency}. Detector for rule:use_css_variables has patterns: {css, variables, consistency, centralized}. Coverage: $3/4 = 0.75$. Rule threshold: 0.50. Match. Grounded.

Running example: Statement "Use Inter for body text" \rightarrow tokens: {inter, body, text}. "inter" \in blacklist. Immediately classified as ungrounded. AntiPattern violation: antipattern:overused_fonts.

5.4 Layer 2: Type-Driven LLM Verification

Layer 2 activates only when Layer 1 is inconclusive — the statement has partial overlap with detectors but doesn't meet the threshold — and an LLM function is available. It makes at most 3 LLM calls per response.

The key mechanism is the **Type Operator Registry**: the node's `@type` determines which verification question to ask.

Node Type	Verification Question	Classifications
Rule	"Does this statement FOLLOW, VIOLATE, or have NO RELATION to this rule?"	FOLLOW \rightarrow grounded, VIOLATE \rightarrow ungrounded
AntiPattern	"Does this statement USE what the pattern forbids, or is it CLEAN?"	VIOLATE \rightarrow ungrounded, CLEAN \rightarrow grounded

Node Type	Verification Question	Classifications
Technique	"Does this statement APPLY or REFERENCE this technique?"	APPLY \rightarrow grounded, REFERENCE \rightarrow grounded

Concepts, Tools, and Traits do not receive Layer 2 verification — they are handled by Layer 1's lower thresholds or classified as persona.

Each prompt includes a **generosity guard**: "If the statement is merely good advice but not explicitly supported by this node, classify as UNRELATED." This prevents the LLM from inflating compliance scores through generous interpretation.

The LLM is forced to output structured JSON: `{"classification": "FOLLOW|VIOLATE|UNRELATED", "reasoning": "..."}.` The first grounded or ungrounded classification terminates the search for that statement. Remaining candidates are skipped.

Running example: Statement "Choose Playfair Display for headlines." Layer 1: no detector exceeds threshold (Playfair Display doesn't appear in any node label). Layer 2 activates. Top candidate: rule:avoid_generic_fonts (partial overlap on "choose" and "fonts" context). LLM prompt: "Does 'Choose Playfair Display for headlines' FOLLOW, VIOLATE, or have NO RELATION to the rule 'Avoid generic fonts like Arial and Inter'?" LLM classifies: FOLLOW (Playfair Display is not a generic font). Statement grounded via type-driven compliance check.

5.5 Layer 3: Edge Policy Traversal

Layer 3 addresses compositional violations — statements that match one node but violate a policy encoded in the edges connecting that node to others. It runs after Layer 1 and Layer 2, only for statements that have been matched to a node.

For each matched node, Layer 3 checks all outgoing verification edges:

avoids: If statement matched node A, and A avoids node B, and $\text{statement_tokens} \cap \text{B.blacklist} \neq \emptyset \rightarrow$ **violation**. The statement addresses the right topic but includes something that topic forbids.

contradicts: If statement matched both node A and node B, and A contradicts B \rightarrow **contradiction**. The statement references two mutually exclusive concepts.

requires: Informational in v1. If statement matched node A, and A requires node B, the system logs whether B appears in the full response. Not scored — future work will incorporate this as a completeness check.

Edge traversal is limited to one hop — source to immediate target. Multi-hop verification is discussed in Section 10 as future work. All traversal is deterministic. Zero LLM calls. Processing time: <0.5ms per statement.

Running example: Statement "For typography, I recommend the clean Inter font." Layer 1 matches concept:typography (tokens {typography} overlap patterns). Layer 3 fires: concept:typography \$\rightarrow\$ avoids \$\rightarrow\$ antipattern:overused_fonts. Target blacklist contains {inter, roboto, arial}. Statement tokens contain "inter". Violation detected via graph path: "Statement addresses 'Typography' but contains 'inter' which is avoided per 'Overused fonts (Inter, Roboto, Arial)'."

This is a violation that neither Layer 1 nor Layer 2 would catch independently. Layer 1 matched the topic (typography). Layer 3 followed the edge and found the forbidden token. The graph structure encodes the policy that connects them.

5.6 Scoring

After all three layers process every statement, the scoring formula from Definition 5 applies:

$$\text{score} = |\text{grounded}| / (|\text{grounded}| + |\text{ungrounded}|)$$

Persona statements — those with no verifiable content in any layer — are excluded from both numerator and denominator. If all statements are persona (no verifiable content detected), the score defaults to 1.0 (all-persona pass).

The default threshold is 0.80. This reflects an 80/20 design principle: at least 80% of the verifiable content must be traceable to the agent's knowledge graph. The remaining 20% allows for legitimate LLM contributions: synthesis, contextual framing, and reasonable inference from provided facts.

When a response scores below the threshold, the system produces:

- 1 The numerical score
- 2 A per-statement breakdown with category, method, and matched node for each statement
- 3 A structured gap list: the specific ungrounded claims and their violation details

The gap list is not just a diagnostic. It is the input specification for the self-education loop (Section 6). Every AEC failure carries the precise blueprint for its own remediation.

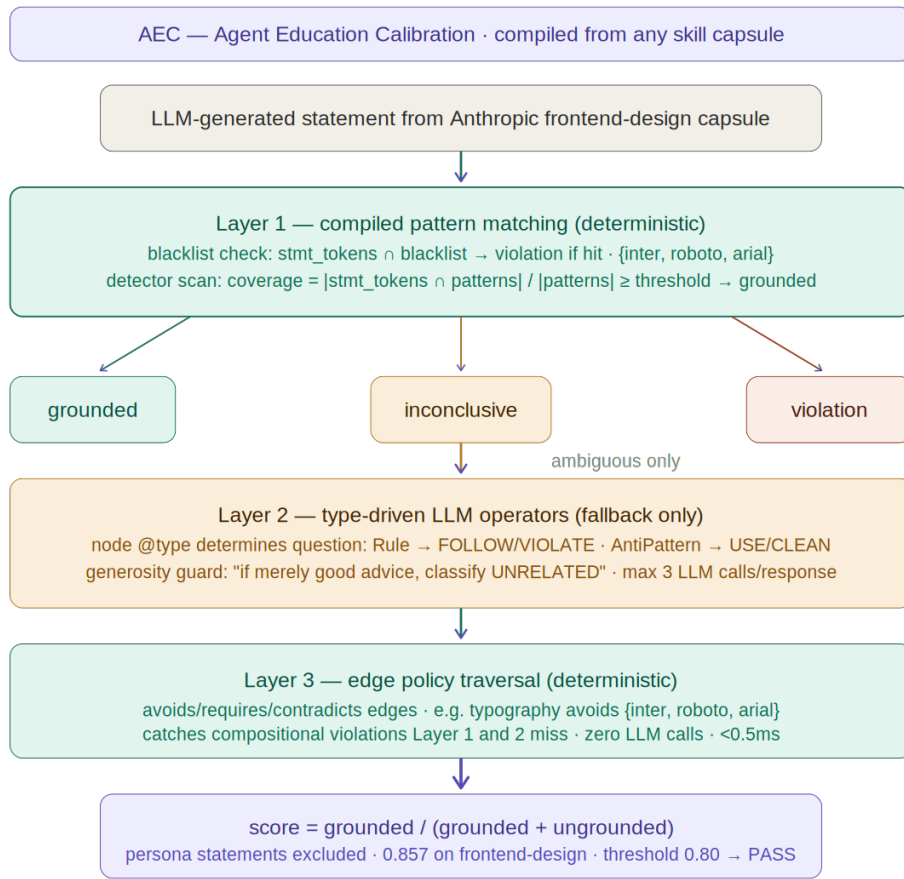


Figure 5: Layers 1 and 3 are fully deterministic. Layer 2 fires only for ambiguous cases.

5.7 Performance

All measurements taken on standard consumer hardware (Apple M-series equivalent, single-threaded Python, no GPU):

Operation	Time	Notes
compile_kg() — 73 nodes	0.62ms	Runs once at capsule load
Layer 1 — per statement	0.1-0.3ms	Set intersection against all detectors
Layer 1 — blacklist check	<0.05ms	Single set intersection
Layer 2 — per statement	2-8 seconds	LLM API call (external latency)
Layer 3 — per statement	<0.5ms	Edge policy traversal
Full verification — 6 statements, no Layer 2	<2ms	Layers 1+3 only
Full verification — 6 statements, with Layer 2	8-20 seconds	Dominated by LLM calls
Distill stage	<0.1ms	Regex only

Operation	Time	Notes
Augment stage	0.1-0.5ms	KB scoring + KG matching
Total framework overhead (excluding LLM)	<3ms	All four DAGR stages

The verification architecture is designed so that the deterministic layers (1 and 3) resolve as many statements as possible before the expensive Layer 2 fires. In practice, Layer 2 activates for 0-3 statements per response. The majority of verification is sub-millisecond.

6. Method: Autonomous Learning

AETHER distinguishes between two distinct failure responses, analogous to the two failure modes of an autonomous robot. **Self-healing** is the immediate response to a verification failure — in standard deployment, the response is delivered with a below-threshold confidence score and the failure is queued for education. In GHOST mode (configurable via `definition.json`), the response is withheld and a structured fault signal is returned instead. The literal `[GHOST]` response token is a designed behavior for high-assurance environments; the score-threshold delivery with confidence annotation is the current default implementation. Either way, the agent signals the fault rather than hiding it. **Self-education** is the longer-term response — the agent researches the specific gaps identified by AEC, validates proposed new knowledge, and integrates verified triples into its knowledge graph so the same failure does not recur.

Self-healing is reactive. Self-education is adaptive. They are sequential, not synonymous. A robot that hits a wall stops (self-healing). It then maps the wall so it doesn't hit it again (self-education). This section describes the self-education mechanism.

6.1 The Education Cycle

When AEC verification produces a score below the threshold (default 0.80), the failure is not simply logged. It is decomposed into a structured curriculum.

The gap list from AEC's Retrieval stage contains the specific ungrounded statements — claims the agent made that could not be verified against any node in the compiled knowledge graph. Each gap entry includes the statement text, the verification method that failed, and the reason for failure (no match, value mismatch, or blacklist violation). These gaps are not vague indicators of poor performance. They are precise specifications of what the agent does not know.

The education cycle proceeds through six stages:

1. QUEUE - AEC failure enters the education queue with query, response, score, and structured gap list
2. RESEARCH - LLM researches the specific gaps, producing proposed knowledge as (subject, predicate, object) triples
3. VALIDATE - Each proposed triple is verified through AEC at a lower threshold (0.5) to ensure internal consistency
4. CONTRADICT - Contradiction gate checks each validated triple against core nodes (Section 6.2)
5. INTEGRATE - Surviving triples are added to the KG with origin "acquired"
6. RE-EVALUATE - Original query re-run against expanded KG; new score computed

Status progression: Each education record moves through: pending \rightarrow researching \rightarrow validated \rightarrow integrated, or diverges to failed (research produced no usable triples) or rejected_contradiction (all triples conflicted with core knowledge).

Running example: The Thomas Jefferson Scholar Agent receives the query "What happened during the Louisiana Purchase and how much did it cost?" The KG contains a single Louisiana Purchase node with two facts: year 1803 and cost \$15 million. The LLM generates a detailed response covering Napoleon, Robert Livingston, James Monroe, the \$11.25 million direct payment, \$3.75 million in assumed debts, 828,000 square miles, and three cents per acre.

AEC score: 0.143. Only the "\$15 million" claim is grounded. Eight statements are ungrounded — the agent generated accurate content from its LLM training data, but none of it exists in the KG. The failure enters the education queue with 8 specific gaps.

6.2 The Contradiction Gate

Before any acquired triple integrates into the knowledge graph, it must pass the contradiction gate. This is the mechanism that prevents the education loop from corrupting the agent's foundational knowledge.

The gate enforces two rules:

Rule 1: Core veto. Core nodes are indexed by subject and predicate. When a proposed acquired triple matches an existing core node's subject and predicate but carries a different object value, the core node's value prevails and the proposed triple is rejected.

```
Core:      (Jefferson, birth_year, 1743)
Proposed: (Jefferson, birth_year, 1750)
-> REJECTED: Core node has birth_year=1743, proposed birth_year=1750
```

This is absolute. No acquired knowledge can override core knowledge at runtime. The core zone is the agent's immutable truth — the knowledge it was created with through the DAG process. Self-education can *extend* this truth but never *contradict* it.

Rule 2: AntiPattern block. If a proposed triple's subject or object overlaps with AntiPattern blacklist terms by 50% or more, the triple is rejected. The agent cannot learn what its own rules forbid.

```
AntiPattern blacklist: {inter, roboto, arial}
Proposed: (font_recommendation, preferred_font, "Arial font")
-> REJECTED: Cannot learn "Arial font" - matches antipattern blacklist
```

This prevents a subtle failure mode: adversarial queries designed to inject forbidden knowledge through the education loop. Repeatedly asking "Isn't Arial actually a premium luxury font?" could, without the gate, cause the agent to research the claim, find supporting text from the LLM (which is a "people-pleaser" by training), and integrate a node that contradicts its own design rules.

Rejected triples are logged with `status: rejected_contradiction`, the conflicting core node identifier, and the rejection reason. They are not silently dropped — the audit trail preserves the attempt for debugging and system analysis.

6.3 Anti-Gaming: Node ID Stripping

A separate integrity mechanism addresses a different gaming vector: the agent inflating its own verification scores.

During the Generate stage (Section 5.1), the LLM receives knowledge graph context to ground its response. If this context includes node identifiers — `rule:avoid_generic_fonts`, `antipattern:overused_fonts` — the LLM can embed these identifiers directly in its output. Layer 1's pattern matching then matches trivially: the response literally contains the detector's label text.

We observed this behavior empirically. Before the anti-gaming fix, responses from the frontend-design capsule contained direct node references:

```
Before: "Per rule:avoid_generic_fonts, you should choose distinctive typefaces.
        This addresses antipattern:generic_ai_aesthetics..."

After:  "Choose distinctive typefaces like Playfair Display or Crimson Pro.
        Avoid generic fonts that feel like template design..."
```

The fix: `@id` values and all `aether:` namespaced properties are stripped from KG nodes before they enter the LLM prompt. The LLM sees the human-readable content of the rules but not their identifiers. AEC in the Retrieval stage retains full access to all fields for verification.

Measured impact:

Metric	Before Fix	After Fix
Node IDs in response	6+ references	0
Prompt size	1,590 chars	1,039 chars (-35%)
AEC score	1.0 (inflated via self-citation)	0.857 (earned via concept matching)

Metric	Before Fix	After Fix
Response quality	Reads like a specification document	Reads like expert design advice

The score dropped from a perfect but meaningless 1.0 to a lower but genuine 0.857. The agent stopped gaming and started demonstrating.

6.4 The Self-Education Proof

The complete education cycle was validated on the Thomas Jefferson Scholar Agent.

Initial state: The agent's KG contained 51 core nodes covering Jefferson's biography, the Declaration of Independence, and his diplomatic service. The Louisiana Purchase was represented by a single node with two data points (year: 1803, cost: \$15 million).

The failure: Query: "What happened during the Louisiana Purchase and how much did it cost?" AEC score: 0.143. One statement grounded (\$15 million cost matched). Eight statements ungrounded. One statement persona. The failure entered the education queue with 8 specific gaps.

The education: The LLM researched the 8 gaps, producing 22 proposed triples covering Napoleon Bonaparte, Robert Livingston, James Monroe, the negotiation timeline, the territory size, the payment structure, and the constitutional questions. AEC validated each at threshold 0.5. The contradiction gate checked each against core nodes — no conflicts found (Louisiana Purchase was under-represented in core, not contradicted). 17 of 22 triples survived validation and contradiction checking. 5 were rejected: 3 failed AEC validation (imprecise formulations), 2 were duplicates of existing core knowledge.

The integration: 17 new triples added to the KG with `origin: "acquired"`. The knowledge graph grew from 51 to 68 nodes.

The re-evaluation: Same query re-run against the expanded KG. AEC score: 0.889. The eight previously ungrounded statements now matched acquired nodes covering Napoleon, Livingston, Monroe, territory size, and payment structure. One statement remained ungrounded (a synthesis claim about constitutional precedent that the education research did not cover precisely enough).

Metric	Before Education	After Education
AEC Score	0.143	0.889
Grounded statements	1	8
Ungrounded statements	8	1
KG nodes	51	68
Triples acquired	—	17
Human intervention	—	Zero

The skill improved through practice. The agent encountered a question it could not adequately answer. AEC identified exactly what was missing. The education loop researched exactly those gaps. The contradiction gate ensured the new knowledge was consistent with the existing core. The agent is now permanently more capable on this topic. The next query about the Louisiana Purchase will be grounded.

The model did not change. **The skill did.**

6.5 Education as Curriculum Generation

The self-education loop inverts the traditional relationship between failure and improvement. In conventional agent systems, a failure is an error to be logged. In AETHER, a failure is a curriculum to be executed.

The gap list produced by AEC's Retrieval stage is not a list of complaints. It is a research specification: "These are the exact facts missing from the knowledge graph. Research them. Validate them. Integrate them." The specificity of the gaps — tied to particular statements, particular nodes, particular verification methods — means the education loop does not wander. It researches exactly what is needed. Nothing more.

This creates a flywheel:

```
Query -> Response -> AEC -> Failure -> Gaps -> Education -> New Knowledge ->
-> Recompile -> Better Verification -> Next Query Benefits
```

Every failure makes the agent more capable. Every capability makes subsequent failures less likely. The knowledge graph grows through use. The compiled verification engine rebuilds at each load to incorporate new detectors from acquired nodes. The agent that has processed 100 queries has more knowledge, more verification coverage, and fewer gaps than the agent that has processed 10.

No fine-tuning. No retraining. No human intervention during the cycle. The model is interchangeable. The skill compounds.

7. System Architecture

This section describes the system-level components that support the DAG creation process and DAGR runtime pipeline: the capsule registry, the orchestrator, dynamic skill creation, and a brief overview of the Ψ (Psi) projection layer for UI actuation.

7.1 The Habitat: Capsule Registry and Routing

The Habitat is AETHER's capsule discovery and routing system. It scans a directory for valid capsule folders, indexes their capabilities from their definition files, and routes incoming queries

to the most appropriate capsule.

Discovery is filesystem-based. Any directory containing the five required capsule files (manifest, definition, persona, KB, KG) is registered as an available agent. Adding a new agent means adding a folder. Removing an agent means removing a folder. No database, no configuration file, no restart required.

Routing uses a weighted scoring algorithm against five signals:

Signal	Weight	Source
Trigger text overlap	0.25	definition.json trigger_text field
Authoritative domain overlap	0.25	definition.json domain_boundaries.authoritative
KG label overlap	0.30	All <code>rdfs:label</code> values across the capsule's KG nodes
Capsule name overlap	0.10	Manifest name field
KG node count (normalized)	0.10	Graph size as tiebreaker

Query tokens are compared against each signal using the same `tokenize()` function from AEC Layer 1, ensuring consistency between routing and verification vocabularies.

Gap detection activates when the highest-scoring capsule falls below a threshold of 0.15. This indicates no capsule in the registry adequately covers the query's topic. The gap is reported with the closest match and its score, providing a structured signal for dynamic skill creation (Section 7.3). Description-based semantic routing — using the capsule's `definition.json` description field as a secondary signal — is planned as a future enhancement (Section 10.2).

7.2 The Orchestrator Capsule

The orchestrator is itself a capsule — a 5-file directory with `agent_type: "router"`. It has a manifest, a definition, a persona (operational, precise, zero embellishment), a KB describing routing methodology, and a KG that is populated dynamically from the registry at load time.

The orchestrator's Generate and Review stages are disabled. It does not generate content. It does not verify content. It routes queries to specialist capsules and returns their results. The user asks a question. The orchestrator determines who should answer. The specialist runs its full DAGR pipeline including AEC verification. The verified response returns through the orchestrator.

This design means the orchestrator eats its own cooking — it is built using the same capsule format it routes to, demonstrating that the format is universal enough to represent both content agents and infrastructure agents.

Routing in practice:

```
# User doesn't need to know which agent exists
python cli.py orchestrate "When was Jefferson born?" -registry ./examples

# Output:
# Routing: "When was Jefferson born?"
#   Candidates:
#     Thomas Jefferson Scholar Agent   score=0.21 * SELECTED
#     doc-coauthoring                  score=0.19
#     ...
#   Dispatching to: Thomas Jefferson Scholar Agent
#
#   Thomas Jefferson was born on April 13, 1743...
#   AEC Score: 1.00 (threshold: 0.8) - PASS
```

Tested routing across three agent categories:

Query	Routed To	Category	Score
"When was Jefferson born?"	jefferson	Scholar	0.21
"What is our strategic vision?"	CEO	Executive Advisor	0.30
"How should I approach typography?"	frontend-design	Skill Agent	0.23
"Configure a Kubernetes cluster?"	GAP DETECTED	—	<0.15

The Kubernetes query triggers gap detection — no capsule in the registry covers this topic. This gap signal is the input for dynamic skill creation.

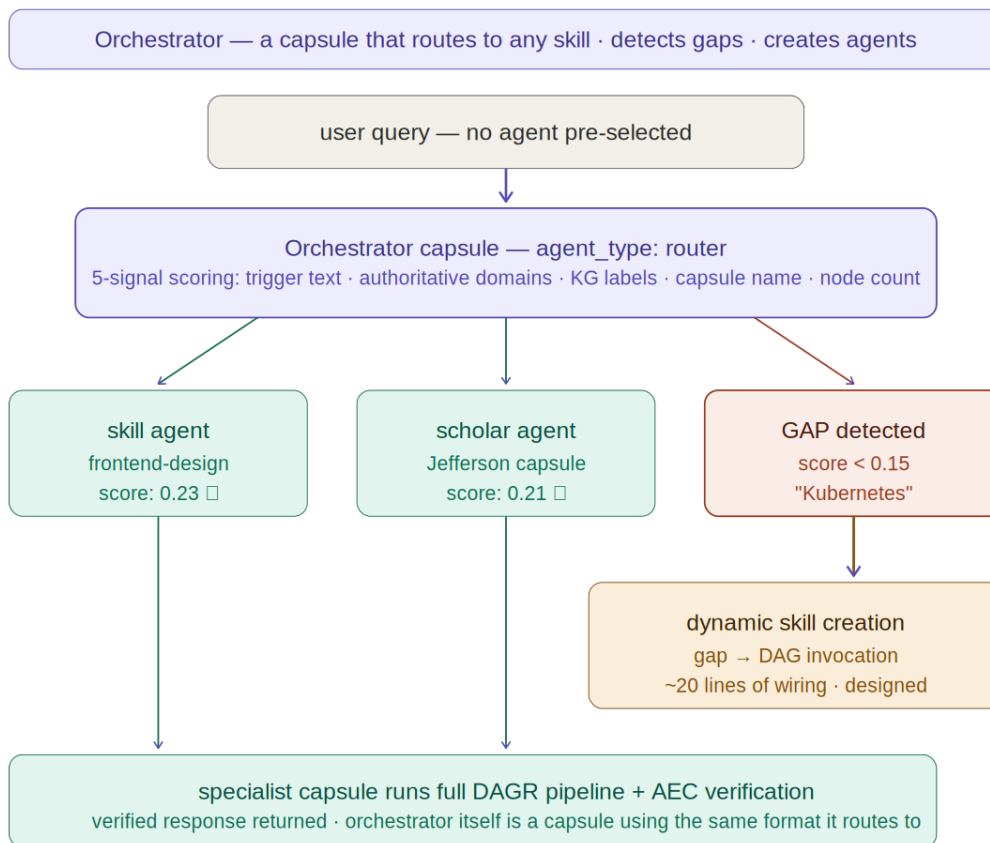


Figure 9: The orchestrator eats its own cooking — built with the same capsule format it routes to. 33 capsules in registry · any skill · any platform.

7.3 Dynamic Skill Creation

When the orchestrator detects a gap, the system has the architectural components to create a new agent autonomously:

```

Gap detected ("Kubernetes" - no capsule matches)
-> Intent captured (D(Q) from Distill provides entities and domain)
-> Research (same LLM mechanism as education loop)
-> Extract (K_graph and K_text from DAG's distillation function)
-> Stamp (stamper generates capsule with UID and lineage)
-> Register (Habitat discovers the new folder)
-> Route (next query on this topic reaches the new capsule)
-> Verify (AEC checks the new capsule's output)
-> Educate (failures improve the new capsule)
-> Mature (after N education cycles, capsule reaches production quality)
  
```

Each step in this chain is independently implemented and tested. The gap detection is proven. The research mechanism is proven (education loop uses it). The extraction skills are proven (DAG process, Section 4). The stamper is proven. The Habitat discovery is proven. AEC verification is proven. The education loop is proven.

The chain connecting gap detection to automatic capsule creation — the trigger that fires the DAG process from an orchestrator gap signal — is the designed but unbuilt link. It represents

approximately 20 lines of code connecting two proven subsystems. The architectural capability exists; the wiring does not yet.

This is the self-creating system: agents that detect what is missing and build what is needed. The orchestrator identifies the gap. DAG creates the agent. DAGR verifies the agent. The education loop improves the agent. The skill emerges from the need.

7.4 Feature Brick Architecture

The capsule's 5-file format enables compositional agent creation. Each file is an independently swappable component — a feature brick:

Brick	What It Carries	Swap Scenario
Persona	Voice, tone, reasoning style	Same knowledge, different personality
KB	Domain expertise (markdown)	Same personality, different domain
KG	Verification rules, typed policies	Same KB, different compliance requirements
Definition	Pipeline config, triggers, thresholds	Same agent, different operational parameters
Manifest	Identity, lineage	Re-stamp for new version

A new agent can be composed by selecting bricks from existing agents: the CEO persona + the enterprise KB + the compliance KG + a custom definition = a specialized compliance advisor capsule. The stamper generates the new manifest with lineage tracking both source capsules.

This compositional model is architectural — enabled by the file-based design — but the tooling for browsing, selecting, and assembling bricks (a visual catalog) is not yet built.

7.5 Capsule Inventory

The current implementation includes 33 capsules across six categories:

Category	Count	Examples	KG Nodes
Scholars	2	Thomas Jefferson (51 nodes), Warren Buffett (48 nodes)	Rich factual graphs

Category	Count	Examples	KG Nodes
Skill Agents	17	frontend-design (73 nodes), brand-guidelines, claude-api, doc-coauthoring, skill-creator, mcp-builder, and 11 additional Anthropic skills	Typed Rules/AntiPatterns /Techniques
Validators	2	aether-validator, test-agent	Verification-focused
Executive Advisors	8	CEO (55 nodes), CTO, CPO, CFO, CISO, CLO, Lead Dev, Lead Data Architect	Domain advisory
Domain Experts	3	Domain Agent Builder, Domain Expert CAP, Agent Performance Manager	Platform-specific knowledge
Infrastructure	1	Orchestrator (router)	Capability graph

All capsules were created through DAG (Section 4) — 17 from Anthropic's official SKILL.md files, 8 from Gemini-generated executive research with AETHER extraction skills, and 8 from manual construction during framework development. All operate through the same DAGR pipeline and AEC verification. Capsules operate independently; multi-capsule interaction is mediated exclusively through the orchestrator.

7.6 The Ψ (Psi) Projection Layer

Current UI approaches require the agent to return structured JSON which a frontend framework then interprets, re-renders, and hydrates — React's hydration process introduces latency between agent state change and visual update, Angular's change detection adds overhead, and traditional Agent-to-UI (A2U) architectures require bespoke integration code for every design system. AETHER's projection layer eliminates this entirely.

Rather than returning data for a framework to render, the agent projects its cognitive state directly into the interface through four integrated components:

DAI Pulse (Dynamic Adaptive Intelligence Pulse). A four-phase state machine mapped to DAGR pipeline stages: *reflex* (Distill complete), *deliberation* (Augment and Generate running), *complete* (AEC passed, full content), and *ghost* (AEC failed, graceful degradation). DAI Pulse emits living impulses — simultaneously driving visual projection, user behavioral signals, analytics events, and user journey markers through a single SSE stream. The agent's cognitive state IS the analytics event. No separate instrumentation required.

CSS Variable Patch (CVP) Protocol. The agent emits semantic CSS custom property mutations via Server-Sent Events: `-aether-state: deliberation`, `-aether-confidence: 0.85`, `-aether-score: 0.857`. Any system that supports CSS custom properties — every modern browser, every design system — can consume them natively. No hydration. No re-render cycle.

EDS Slivers (Embodied Design Slivers). Pre-existing HTML fragments under 1KB, bound to capsules via namespace attributes, containing zero JavaScript. Slivers do not need to be built for the agent — they already exist in the design system. The agent inhabits them. When DAI Pulse emits a state change, slivers respond through standard CSS inheritance.

2KB Edge Orchestrator. A single JavaScript file ($\leq 2,000$ bytes, 896 bytes gzipped) loaded at the browser edge. It discovers slivers, connects to CVP streams via SSE, and applies CSS variables to targeted elements. Content is injected via `data-aether-slot` attributes using `textContent` (no `innerHTML`, no XSS vector).

The projection layer is design-system agnostic — React, plain HTML, enterprise design systems, any system supporting CSS custom properties. An adapter pattern enables snap-in integration with existing systems or full replacement. Agents no longer need UI built for them. They inhabit UI that already exists.

The Ψ layer is implemented and validated. A full treatment of the projection architecture — including intent pub/sub, multi-agent cognitive scoping, behavioral analytics integration, and enterprise design system adapters — is the subject of a companion paper.

7.7 Agent-to-Agent Communication via KG Subgraph Exchange

Current multi-agent systems pass unverified text between agents. A response that hallucinated in one agent travels to the next agent as grounded fact. Verification quality degrades with every communication hop.

AETHER's A2A communication model inverts this. When agents communicate, they exchange typed JSON-LD KG subgraph fragments — not raw text. Each receiving agent runs AEC on incoming knowledge before integrating it. The verification standard does not degrade across hops because every agent in the network applies the same compiled verification to everything it receives.

The protocol mirrors the robot team communication model: agents communicate via a standardized protocol, every message is typed and structured, and every receiver independently validates what it receives. An agent cannot be poisoned by a hallucinating peer because AEC's contradiction gate blocks incoming knowledge that conflicts with the receiving agent's core nodes.

The JSON-LD format used by AETHER capsules enables direct interoperability with enterprise A2A communication protocols without format translation. This positions AETHER as a verification layer that can operate within enterprise multi-agent architectures without requiring protocol changes.

Multi-capsule evaluation — measuring score propagation, latency accumulation, and error compounding across agent networks — is ongoing. Single-capsule verification is proven. Network-level verification is the designed next step.

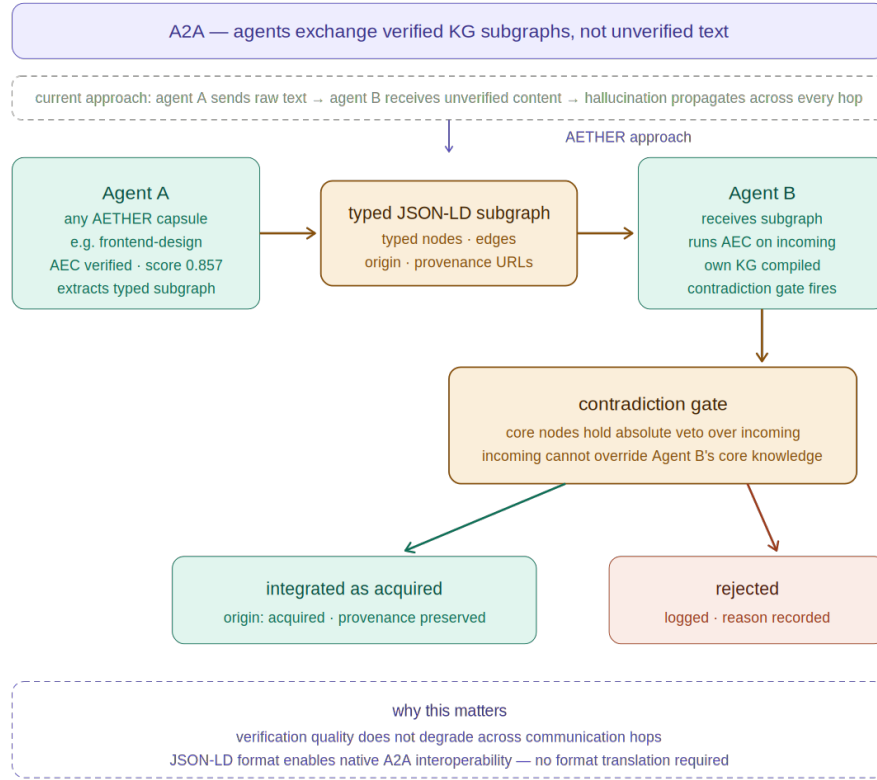


Figure 8: Every receiving agent independently verifies what it receives.
An agent cannot be poisoned by a hallucinating peer.
Agents do not pass text. They pass typed, verified knowledge.

8. Experiments

This section presents experimental results across three evaluation dimensions: factual grounding discrimination, skill verification effectiveness, and self-education improvement. All experiments use the published codebase and can be reproduced using the commands provided.

8.1 Experimental Setup

Hardware: Standard consumer laptop, Apple M-series equivalent, single-threaded Python. No GPU. No vector database. No external services beyond the LLM API.

LLM Provider: Anthropic Claude Sonnet for all generation and Layer 2 verification calls. Temperature: default (model-controlled). No custom sampling parameters.

Capsules Under Test:

- Thomas Jefferson Scholar Agent: 51 core KG nodes, 28,810 bytes KB, 43 graph nodes with 37 triples across 13 entity types.
- Frontend-design Skill Agent: 73 core KG nodes (22 Rules, 22 Techniques, 14 Concepts, 6 AntiPatterns, 5 Traits, 4 Tools), 3,956 bytes KB, 180 triples.
- 8 Executive Advisor capsules: 45-55 KG nodes each, created via DAG from Gemini-generated research.

Evaluation Protocol: Each experiment reports AEC score, per-statement breakdown (grounded/ungrounded/persona), verification method per statement, and timing. LLM responses vary between runs; reported scores represent single runs with the specific response text available in the repository for verification.

8.2 Experiment 1: Factual Grounding Discrimination

Hypothesis: AEC correctly discriminates between fully grounded, partially grounded, and severely ungrounded responses on factual content.

Design: Three queries selected to exercise the full spectrum of knowledge coverage against the Jefferson capsule:

Query	KG Coverage	Rationale
"When was Thomas Jefferson born?"	Full (birth date, location, name all in KG)	All verifiable claims should ground
"Jefferson vs Hamilton disagreements?"	Partial (Hamilton absent from KG)	Jefferson claims ground; Hamilton claims don't
"Louisiana Purchase details and cost?"	Minimal (1 node, 2 facts)	Most claims will be LLM-generated

Results:

Query	AEC Score	G	U	P	Input Tokens	Result
Birth	1.000	4	0	0	847	PASS
Hamilton	0.600	3	2	0	189	FAIL
Louisiana	0.143	1	8	1	46	FAIL

Analysis: The scores correlate directly with input token count — a proxy for how much grounding context the Augment stage found. Full coverage (847 tokens) produces a perfect score. Partial coverage (189 tokens) produces a proportional score — exactly 3 of 5 verifiable claims are grounded, reflecting the fact that Jefferson-related content exists in the KG but Hamilton-related content does not. Minimal coverage (46 tokens) produces a near-zero score — only the "\$15

million" cost claim matched the single Louisiana Purchase node.

Significance: AEC does not produce binary pass/fail. It produces proportional scores that reflect the precise knowledge coverage of the capsule. The gap list from the 0.143 failure (8 specific ungrounded claims about Napoleon, Livingston, Monroe, territory size, and payment structure) becomes the education curriculum validated in Experiment 3.

```
# Reproduce:
python cli.py run examples/jefferson "When was Jefferson born?" -provider anthropic -report full
```

8.3 Experiment 2: Skill Verification Effectiveness

Hypothesis: AEC concept layers produce meaningful verification scores on skill agents where factual AEC alone produces trivial scores.

Design: The frontend-design capsule (73 KG nodes) queried with "How should I approach typography for a luxury brand?" — a query that exercises Rules, Techniques, and AntiPatterns simultaneously.

Condition A — Factual AEC only (concept layers disabled):

Metric	Value
AEC Score	1.000
Grounded	0
Ungrounded	0
Persona	2
Verdict	Trivially perfect — nothing verified

All statements classified as Persona. No numbers, dates, or names to extract. AEC's factual gate has no purchase on skill content. The score is meaningless.

Condition B — Full AEC with concept layers:

Metric	Value
AEC Score	0.857
Grounded	6
Ungrounded	1
Persona	1
Verdict	Meaningful — rules checked, violation caught

Six statements grounded via compiled pattern matching (Layer 1) and type-driven verification (Layer 2) against Rules and Techniques. One statement flagged as ungrounded: anti-pattern

violation (the word "Inter" detected in the compiled blacklist). One statement classified as Persona (genuine opinion with no KG match).

Condition C — Without anti-gaming fix:

Metric	Value
AEC Score	1.000
Grounded	4
Ungrounded	0
Persona	0
Prompt size	1,590 chars
Node IDs in response	6+
Verdict	Inflated — agent self-cited to game verification

The LLM embedded KG node identifiers directly in the response ("per rule:avoid_generic_fonts", "antipattern:generic_ai_aesthetics"). Layer 1 matched trivially. The score was perfect but the verification was meaningless — the agent was citing its own test answers.

Condition D — With anti-gaming fix:

Metric	Value
AEC Score	0.857
Prompt size	1,039 chars (-35%)
Node IDs in response	0
Response quality	Expert design advice, not specification document
Verdict	Earned — genuine concept matching

Compilation Performance:

Metric	Value
Nodes compiled	73
Detectors produced	73
Blacklist tokens	5
Edge policies	12
Compilation time	0.62ms

```
# Reproduce Condition B:
python cli.py run examples/frontend-design-v1.0.0-ff6ab491 \
  "How should I approach typography for a luxury brand?" \
  -provider anthropic -report full
```

8.4 Experiment 3: Self-Education Improvement

Hypothesis: The education loop autonomously improves agent knowledge when AEC detects gaps, without human intervention during the cycle.

Design: The Jefferson capsule's Louisiana Purchase failure from Experiment 1 (score 0.143, 8 ungrounded statements) is used as the education input.

Education Process:

Stage	Metric	Value
Queue	Gaps queued	8 specific ungrounded claims
Research	Triples proposed	22
Validate	Triples passing AEC (threshold 0.5)	19
Contradict	Triples rejected (core conflict)	0
Contradict	Triples rejected (duplicate)	2
Integrate	Triples added to KG	17
KG growth	Nodes before	51
KG growth	Nodes after	68

Re-evaluation:

Metric	Before Education	After Education	Delta
AEC Score	0.143	0.889	+0.746
Grounded	1	8	+7
Ungrounded	8	1	-7
Persona	1	1	0

The single remaining ungrounded statement concerned constitutional precedent implications — a synthesis claim that the education research did not cover with sufficient specificity. The persona statement remained unchanged (interpretive commentary).

Acquired Knowledge Sample:

Subject	Predicate	Object	Origin
Louisiana Purchase	negotiated_by	Robert Livingston, James Monroe	acquired
Louisiana Purchase	seller	Napoleon Bonaparte / France	acquired
Louisiana Purchase	territory_size	approximately 828,000 square miles	acquired
Louisiana Purchase	payment_structure	\$11.25M direct + \$3.75M assumed debts	acquired

Integrity Verification: No acquired triple contradicted any core node. The contradiction gate processed all 22 proposed triples; 0 were rejected for core conflict. 2 were rejected as duplicates of existing core knowledge (year: 1803 and cost: \$15M already existed). 3 failed AEC validation at the 0.5 threshold (imprecise formulations).

Human intervention during the cycle: Zero. The gap list from AEC failure was the only input. The education loop conducted all research, validation, contradiction checking, integration, and re-evaluation autonomously.

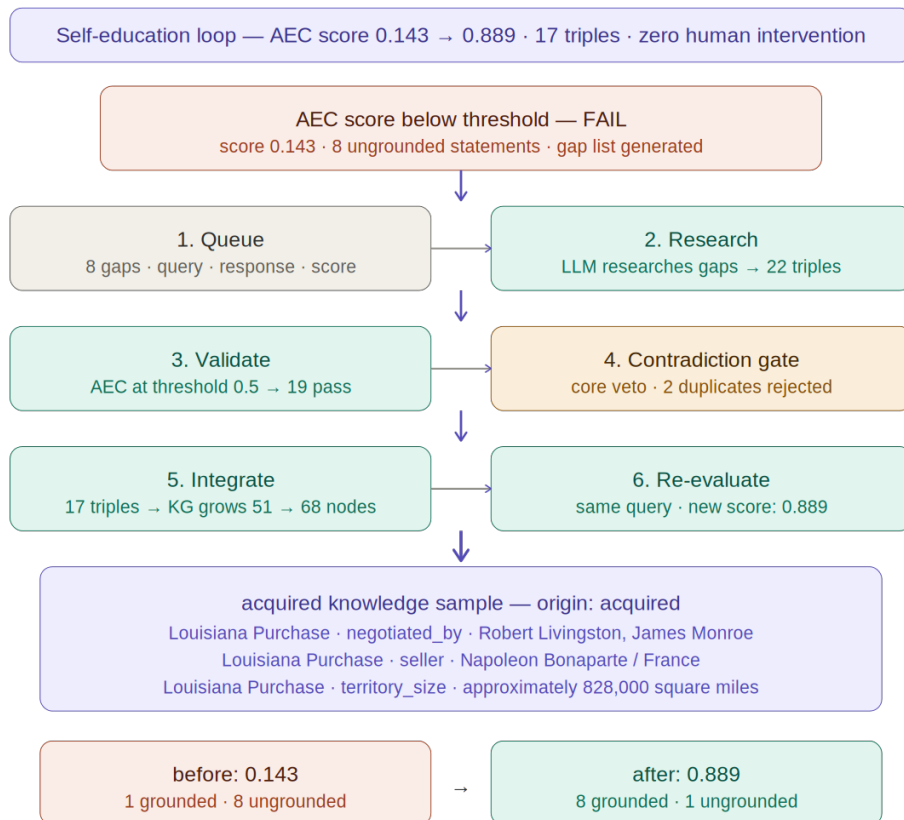


Figure 7: The model did not change. The skill did.
Zero human intervention during the education cycle.

```
# Reproduce (requires initial failure + education run):
python cli.py run examples/jefferson "Louisiana Purchase details and cost?" -provider anthropic
python cli.py educate examples/jefferson -item 0 -provider anthropic
```

8.5 Experiment 4: Orchestrator Routing

Hypothesis: The orchestrator correctly routes queries across three agent categories without human-specified agent selection.

Design: Four queries spanning different domains, routed via the orchestrator capsule against 33 production capsules.

Query	Expected Target	Actual Target	Score	Correct?
"When was Jefferson born?"	jefferson	jefferson	0.21	[OK]
"What is our strategic vision?"	CEO	CEO	0.30	[OK]
"How should I approach typography?"	frontend-design	frontend-design	0.23	[OK]
"Configure a Kubernetes cluster"	GAP	GAP (<0.15)	—	[OK]

All four routing decisions correct. The Kubernetes gap detection demonstrates that the system does not force a bad match — it reports the absence of capability.

```
# Reproduce:
python cli.py orchestrate "When was Jefferson born?" -registry ./examples -dry-run
```

8.6 Framework Performance Summary

Component	Metric	Value
compile_kg() — 73 nodes	Compilation time	0.62ms
Layer 1 — per statement	Verification time	0.1-0.3ms
Layer 1 — blacklist check	Violation detection	<0.05ms
Layer 2 — per LLM call	External latency	2-8 seconds (external LLM API; AETHER overhead is prompt construction only)
Layer 3 — per statement	Edge traversal	<0.5ms
Full DAGR pipeline (excl. LLM)	Framework overhead	<3ms
Orchestrator routing	Query \rightarrow dispatch	<60ms

Component	Metric	Value
Education cycle (1 round)	Full loop	30-90 seconds
Total codebase	Python files	15 files, standard library only
Dependencies	External	anthropic SDK (optional)

9. Discussion and Limitations

9.1 Implications for Agent Accountability

AETHER demonstrates that the verification gap in current agent systems is an architectural choice, not an inherent constraint. The key insight is that knowledge graphs can serve simultaneously as retrieval stores, policy engines, and verification runtimes — the same artifact, compiled once, serving three functions. No existing framework we surveyed exploits this property.

The practical implication is immediate: any system that maintains a knowledge graph for RAG can, with the addition of type annotations and a compilation step, gain deterministic output verification at sub-millisecond cost. The verification does not require a separate infrastructure (no vector database, no judge model, no GPU). It requires only that the knowledge graph's nodes carry types that encode verification semantics.

9.2 The Skill Thesis Revisited

The experimental results support the five-property skill definition introduced in Section 3.1. The Jefferson capsule demonstrated Knowledge (51 KG nodes), Identity (persistent manifest with lineage), Expression (formal academic persona), Behavior (scholar agent type with AEC gates), and Improvement ($0.143 \rightarrow 0.889$ through autonomous education). No property was simulated or mocked — each was exercised through the DAGR pipeline under standard conditions.

The portable proof — that the same capsule produces the same verification results regardless of which LLM generates the response — follows from the architecture: AEC compiles and verifies against the capsule's KG, not against the model's weights. The model is the engine. The skill is the asset. The model did not change. **The skill did.**

9.3 Relationship to Enterprise Architecture

Enterprise AI reference architectures increasingly adopt a multi-layer stack pattern: experience, cognitive, multi-agentic, capability, and backend layers. The cognitive layer typically includes knowledge graph and document grounding components. Content filtering, where present, generally operates on the input side — sanitizing what reaches the AI engine. Output verification is absent from published reference architectures reviewed.

AEC addresses this gap directly. It sits between the AI engine's response and delivery to the user, verifying compliance against the agent's compiled knowledge graph. The JSON-LD format used by AETHER's knowledge graphs enables native interoperability with enterprise A2A communication protocols without format translation.

The broader enterprise implication: any organization that already maintains structured knowledge — ontologies, policy documents, compliance frameworks — can compile that knowledge into AEC verification detectors. The knowledge they already have becomes the verification engine they currently lack.

9.4 Multi-LLM Validation

During the development of AETHER, architectural decisions were independently reviewed by five LLM systems (Gemini, GPT-4, Kimi, Copilot, Grok) across two review rounds. Points of unanimous consensus included: the tiered deterministic-first verification architecture, the type-driven operator approach, the core/acquired immutability distinction, and the self-education loop as a genuine differentiator. This cross-model review process is itself a form of the verification thesis: no single model's opinion governs the architecture.

9.5 Limitations

We identify eight limitations of the current implementation, organized by severity.

Negation blindness (High). The blacklist detector does not understand negation. "Avoid using Inter" triggers a violation because "Inter" is in the blacklist, even though the agent is advising against the anti-pattern. Distinguishing "avoid Inter" from "use Inter" requires natural language understanding beyond tokenization — likely lightweight dependency parsing or negation scope detection. This is the most impactful known limitation.

LLM-dependent scores (High). Because the Generate stage uses an LLM, responses vary between runs. AEC scores for the same query against the same capsule can vary by ± 0.15 depending on the specific response generated. The verification architecture is deterministic; the content being verified is not. Reproducibility requires fixing the response text, which we provide in the repository.

Single-capsule evaluation (Medium). All experimental results are from individual capsules tested in isolation. Multi-capsule interactions — where orchestrated queries traverse multiple agents — have not been evaluated for score propagation, latency accumulation, or error compounding.

Layer 2 scalability (Medium). Layer 2's LLM calls (2-8 seconds each) represent a latency and cost bottleneck in high-throughput environments. The architecture mitigates this through the cascade design — Layer 1 resolves the majority of statements deterministically before Layer 2 fires. However, capsules with thin knowledge graphs (few typed nodes, weak label overlap) will trigger Layer 2 more frequently. The primary mitigation is higher-quality DAG distillation: richer

extraction produces more detectors, which means more Layer 1 matches, which means fewer Layer 2 calls.

Threshold sensitivity (Medium). The type-specific thresholds (Rules 0.50, Techniques 0.55, AntiPatterns 0.40, Concepts 0.30) were tuned empirically against one capsule (frontend-design, 73 nodes). Broader validation across diverse knowledge graphs — different domains, different node counts, different edge densities — is needed to confirm these values generalize. A grid search or learned threshold approach may be warranted.

Augment weakness (Medium). The top-3 KB paragraph matching in the Augment stage uses simple entity overlap, which misses semantically relevant paragraphs that use different terminology. A "Quick Answer" block at the top of each kb.md — a 50-word distilled summary that the Augment stage always retrieves — is a planned mitigation (not yet implemented).

Education learns tokens, not principles (Low). During frontend-design testing, the education loop acquired specific terms (font names, CSS property names) rather than generalizable design principles. The agent learned that "Playfair Display" is acceptable but did not generalize to "serif fonts with high contrast are appropriate for luxury." This suggests that education research prompts may need to request principles, not just facts.

Single-hop edge traversal (Low). Layer 3's edge policy traversal is limited to one hop — source to immediate target. Multi-hop reasoning chains (A avoids B, B requires C, therefore A has implications for C) are not evaluated. Extending to multi-hop introduces combinatorial complexity that must be bounded.

9.6 Threats to Validity

Internal validity. AEC scores are computed by the same system that generated the response. While Layers 1 and 3 are deterministic (no model involvement), Layer 2 uses an LLM call — introducing the same judge-bias concern we critique in Section 2.2. The mitigation is that Layer 2 is fallback only, the generosity guard constrains it, and the majority of statements are resolved by deterministic layers.

External validity. All capsules were created by the authors using the DAG process. No external capsule creators have tested the framework. The extraction skills may encode assumptions about source material structure that do not generalize to arbitrary documents.

Construct validity. The AEC score measures grounding against the capsule's own KG — not against ground truth. A capsule with an incorrect KG will produce high scores on incorrect responses. AEC measures consistency with stated knowledge, not correctness of that knowledge. An external auditor capsule (Section 10) addresses this.

Conclusion validity. The evaluation covers 33 capsules across six categories, providing preliminary evidence across diverse agent types. However, statistical power is limited: larger-scale evaluation across hundreds of capsules, diverse domains, and multiple LLM providers is needed to establish generalizability with confidence.

10. Conclusion, Broader Impact, and Reproducibility

10.1 Summary

This paper presented AETHER, a minimal agent framework derived from a pattern observed in autonomous robot architecture: the robot already had everything an AI agent needs — defined boundaries, compiled intelligence, operational learning, failure recovery, persistent identity, and a communication protocol. AETHER is what happens when you build AI agents the same way.

Each agent is a capsule: five files in a folder, each mapping to a robot subsystem. The knowledge graph is the compiled navigation mesh. The persona is the team identity. The definition is the communication protocol. The self-education loop is the terrain mapping mechanism. Copy the folder. Copy the agent. Point it at any LLM. The model is the motor. The skill is the asset.

Two proposed architectural processes govern the agent lifecycle. **DAG (Distilled Augmented Generation)** creates agents by distilling knowledge from any source — including any platform's skill files — augmenting it with typed relationships, and generating a self-contained capsule. **DAGR (Distillation + Augment + Generation + Retrieval)** runs agents through a four-stage pipeline where the Retrieval stage compiles the knowledge graph into executable policy checkers and verifies every generated response.

Agent Education Calibration (AEC) — the verification engine within DAGR's Retrieval stage — transforms the knowledge graph from a passive data store into an active policy engine. The node's `@type` drives the verification strategy. The edges encode composed policies. Compilation runs once at load time in $O(|N|)$. Verification runs in sub-millisecond time via set intersection. No embeddings. No vector databases. No GPU.

When verification fails, the self-education loop identifies the specific knowledge gaps, researches them via LLM, validates proposed knowledge through AEC itself, enforces a contradiction gate where immutable core knowledge holds absolute veto, and integrates verified triples. The agent that fails is the agent that learns. The failure specifies its own curriculum.

The model did not change. **The skill did.**

10.2 Future Work

Near-term (designed, not built):

Negation detection. Distinguishing "avoid Inter" from "use Inter" in the context of anti-pattern blacklists. Likely requires lightweight dependency parsing or negation scope detection beyond tokenization.

Dynamic skill creation trigger. Connecting gap detection (proven) to automatic DAG invocation (proven) — the ~20 lines of wiring that enable fully autonomous agent creation from detected

need.

External auditor capsule. A separate agent whose sole function is skeptical verification of other agents' output, using a different persona, different knowledge graph, and potentially a different LLM — breaking the self-grading concern identified in Section 9.6.

A2A subgraph exchange. Implemented as an architectural design in Section 7.7. When agents communicate, they share typed, AEC-verifiable KG fragments rather than unstructured text. Each receiving agent verifies incoming knowledge against its own compiled policies before integration — creating trust chains where verification quality does not degrade with communication hops. Multi-capsule network evaluation is the near-term validation target.

Quick Answer block in kb.md. A 50-word distilled summary at the top of each knowledge base that the Augment stage always retrieves, addressing the augment weakness identified in Section 9.5.

Medium-term (research directions):

Knowledge decay. Adding `utility_score` and `last_accessed` fields to acquired nodes with temporal decay at compile time. Nodes below a utility threshold are deprecated and excluded from compiled detectors — preventing unbounded KG growth.

Multi-hop edge traversal. Extending Layer 3 from one-hop to bounded multi-hop verification using type-constrained path enumeration. Legal but unverified paths classified as hypotheses and queued for education.

Benchmark comparison. Formal evaluation of AEC against RAGAS faithfulness, TruLens guardrails, and DeepEval metrics on a standardized dataset. This paper reports AEC results in isolation; comparative evaluation would strengthen the positioning.

Cross-model generalization. Systematic evaluation of the same capsules across Claude, GPT-4, Gemini, and open-source models to quantify the portability claim.

10.3 Broader Impact

AETHER's verification architecture has implications beyond the immediate agent framework context.

For enterprise AI governance: Organizations deploying AI agents in regulated industries (finance, healthcare, legal) require audit trails for every generated response. AEC's per-statement verdicts with matched node evidence provide exactly this — a complete, deterministic record of what the agent knew, what it verified, and what it could not ground. In the context of the EU AI Act, AEC's per-statement audit trail directly supports the transparency requirements of Article 13.

For knowledge currency: Provenance nodes with live URLs and scheduled validation sweeps address a failure mode that affects every static knowledge system — knowledge that was true

when written may no longer be true. AETHER's living knowledge graph detects dead sources, flags stale nodes, and triggers the self-education loop automatically when source content changes. The agent's knowledge has a biography, not just a value.

For AI safety research: The contradiction gate and anti-gaming mechanisms demonstrate that structural integrity can be enforced without relying on the model's cooperation. The agent cannot corrupt its own rules because the rules compile from immutable sources, and the agent never sees the compilation parameters. This is safety through architecture, not through training.

For the open-source agent ecosystem: The 280,000+ skills published across Claude Code, Copilot, and other platforms currently have no verification mechanism. AETHER's DAG process can ingest any SKILL.md and produce a self-verifying capsule — adding accountability to the existing skill ecosystem without requiring changes to the platforms that host those skills.

For any domain, any scale: The capsule pattern generalizes beyond enterprise software. Any object, any domain, any knowledge corpus can become a verified, self-educating agent. A vehicle that knows its own service history. A building that knows its systems. A course of study that knows what the student doesn't know yet. The five-file format is the minimum viable embodiment of expertise — portable, verifiable, self-improving — regardless of domain or deployment context.

Potential risks: A verification system that produces high scores could create false confidence if the underlying knowledge graph is incorrect. AEC measures consistency with the capsule's own knowledge, not ground truth. Organizations deploying AEC should ensure that core KG nodes are reviewed by domain experts before production use. The external auditor capsule (Section 10.2) and provenance URL validation are designed to mitigate this risk.

10.4 Reproducibility Statement

All experimental results reported in this paper can be reproduced using the public repository:

Repository: github.com/jeff0926/aether

Requirements: Python 3.11+, standard library only. The `anthropic` SDK is required only for LLM-dependent experiments (Generate stage and Layer 2 verification). AEC compilation, Layer 1 verification, Layer 3 edge traversal, and orchestrator routing run without any external dependency.

Reproducibility commands:


```
# Factual grounding (Experiment 1)
python cli.py run examples/jefferson "When was Jefferson born?" -provider anthropic -report full
python cli.py run examples/jefferson "Jefferson vs Hamilton disagreements?" -provider anthropic -report full
python cli.py run examples/jefferson "Louisiana Purchase details and cost?" -provider anthropic -report full

# Skill verification (Experiment 2)
python cli.py run examples/frontend-design-v1.0.0-ff6ab491 \
    "How should I approach typography for a luxury brand?" -provider anthropic -report full

# Self-education (Experiment 3)
python cli.py educate examples/jefferson -item 0 -provider anthropic

# Orchestrator routing (Experiment 4)
python cli.py orchestrate "When was Jefferson born?" -registry ./examples -dry-run

# AEC standalone verification (no LLM required)
python cli.py verify "Thomas Jefferson was born in 1743" -r examples/jefferson/jefferson-kg.jsonld
python cli.py verify "Use Inter for body text" \
    -r examples/frontend-design-v1.0.0-ff6ab491/frontend-design-v1.0.0-ff6ab491-kg.jsonld
```

Note on LLM variability: Because the Generate stage uses an LLM, exact AEC scores will vary between runs as the model produces different responses. The verification architecture is deterministic — the same response text against the same compiled KG will always produce the same score. Reference response texts for all reported scores are available in the repository.

Capsule corpus: 33 capsules are included in the repository's `examples/` directory, spanning factual scholars, procedural skill agents, executive advisors, validators, domain experts, and the orchestrator. All were created through the DAG process described in Section 4.

References

- 1 Lewis, P., et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." NeurIPS 2020.
- 2 Guu, K., et al. "REALM: Retrieval-Augmented Language Model Pre-Training." ICML 2020.
- 3 Es, S., et al. "RAGAS: Automated Evaluation of Retrieval Augmented Generation." arXiv:2309.15217, 2023.
- 4 Liu, Y., et al. "G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment." arXiv:2303.16634, 2023.
- 5 Zheng, L., et al. "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena." NeurIPS 2023.
- 6 Chase, H. "LangChain." GitHub, 2022.
- 7 Wu, Q., et al. "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation." arXiv:2308.08155, 2023.
- 8 Microsoft. "Semantic Kernel." GitHub, 2023.
- 9 Joao, M., et al. "CrewAI: Framework for Orchestrating Role-Playing AI Agents." GitHub, 2024.

- 10 Bai, Y., et al. "Constitutional AI: Harmlessness from AI Feedback." arXiv:2212.08073, 2022.
- 11 Ouyang, L., et al. "Training Language Models to Follow Instructions with Human Feedback." NeurIPS 2022.
- 12 Wang, G., et al. "Voyager: An Open-Ended Embodied Agent with Large Language Models." arXiv:2305.16291, 2023.
- 13 Hu, S., et al. "Automated Design of Agentic Systems." arXiv:2408.08435, 2024.
- 14 Yao, S., et al. "ReAct: Synergizing Reasoning and Acting in Language Models." ICLR 2023.
- 15 Sun, H., et al. "AdaPlanner: Adaptive Planning from Feedback with Language Models." NeurIPS 2023.
- 16 Knublauch, H. & Kontokostas, D. "Shapes Constraint Language (SHACL)." W3C Recommendation, 2017.
- 17 Bordes, A., et al. "Translating Embeddings for Modeling Multi-relational Data." NeurIPS 2013.
- 18 Sun, Z., et al. "RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space." ICLR 2019.
- 19 W3C. "JSON-LD 1.1." W3C Recommendation, 2020.
- 20 Bi, S., et al. "Automating Skill Acquisition through Large-Scale Mining of Open-Source Agentic Repositories." arXiv:2603.11808, March 2026.
- 21 Steinberger, P. "SoulSpec: The Open Standard for AI Agent Personas." soulspec.org, 2025.
- 22 Kim, S., et al. "FactKG: Fact Verification via Reasoning on Knowledge Graphs." ACL 2023.

AETHER — Adaptive Embodied Thinking — Holistic Evolutionary Runtime 864 Zeros LLC — March 2026 github.com/jeff0926/aether