The Grounding Gate: Verified Tool Selection for AI-Driven Research

Thomas Dionysopoulos, CFA

Abstract

When an AI system selects a computational tool, its proposal is an *assertion*: "this tool is appropriate." Without verification, the assertion flows directly into execution. The tool may be structurally valid—it exists, it type-checks—but the user's request does not justify it. We call this the *assertion gap*: the distance between a tool selection that is valid and one that is verified.

We present a *grounding gate* that closes this gap through *evidence-carrying tool selection*. Each tool call carries explicit evidence—match mode, confidence score, and a cryptographic capability hash—linking the selection to the user's terms. A deterministic verification function checks this evidence before execution; proposals lacking evidence are rejected even if they name real capabilities. The architecture enforces a **wall property**: no unverified tool selection reaches execution. Evidence stability is ensured by a behavior-derived identity model where discovery metadata is excluded from capability hashes by construction, so the registry can improve discoverability without invalidating prior evidence or execution hashes.

We evaluate on 30 prompts across 5 categories (4 strategy families, 9 metrics, 36 valid combinations). An assertion-only pipeline (schema validation, no verification) executes unwarranted capabilities at **23.3%**; the verified pipeline reduces this to **10.0%** (Fisher exact $p = 0.027$), eliminating them entirely on undiscoverable prompts (100%→0%). On adversarial prompts exploiting discovery-alias gaps, the verified pipeline has a higher failure rate—a tradeoff inherent to constraining the admissible set. Repeated executions produce bit-identical hashes across all 50 runs; an 8-layer execution hash decomposes provenance for fault localization without re-execution. Verification overhead is under 14 ms.

1 Introduction

When an AI system selects a computational tool, its selection is an *assertion*: "this tool is appropriate for the user's request." In current architectures, assertions flow directly into execution. The tool exists. The output type-checks. But no mechanism verifies that the user's request *justifies* the selection. We call the distance between a structurally valid selection and a verified one the **assertion gap**.

Consider a researcher who prompts: *"Build a momentum strategy on equity futures, ranked by Sharpe ratio."* A standard AI pipeline passes this to an LLM, which proposes `MOM_REV` (mean-reversion) as the strategy family and `SRP` as the ranking metric. Both are valid: `MOM_REV` is a registered strategy family; `SRP` computes the Sharpe ratio. Schema validation confirms the proposal is well-formed. Execution proceeds—and produces a mean-reversion strategy: the *opposite* computational signal from what the researcher requested. The two families produce computationally opposite outputs from the same infrastructure. *The output is correct in form and exactly wrong in substance.*

This failure is invisible to schema validation, constrained decoding, or any check that operates on the set of valid capabilities without reference to the user's request. The proposed names are valid. The pipeline is well-typed. The output is structurally correct. The problem is that the selection was an unverified assertion, and the assertion was wrong.

The root cause is architectural: tool selection is treated as an assertion that requires no evidence. The LLM's output flows directly into the execution engine, constrained only by structural checks on its format (Figure 1, top row). Every tool-augmented LLM framework we are aware of—ReAct [1], Toolformer [2], LangChain [3]—shares this property: the selection is asserted, not verified.

We present an architecture that converts tool selection from assertion to **verified claim** through a *grounding gate*. The system follows a three-stage pipeline: *discover* which capabilities match the user's natural-language terms by querying a live registry (236 capabilities), *verify* that every name in the AI's

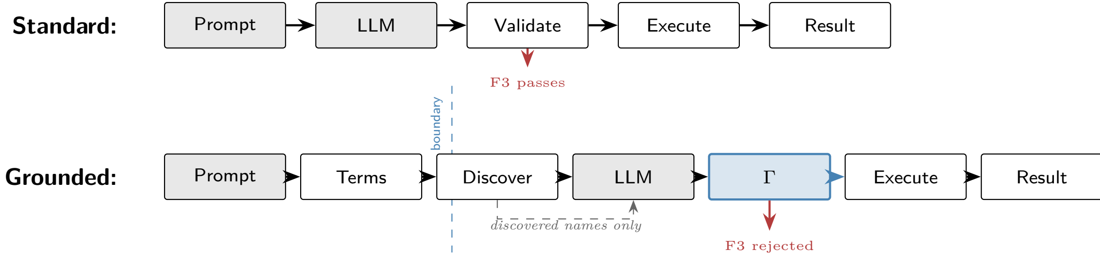

Figure 1: Pipeline comparison. Top: standard pipeline—the LLM proposes directly, validation passes F3 (valid-but-unwarranted). Bottom: grounded pipeline—discovery constrains the LLM's vocabulary, grounding gate Γ rejects unwarranted proposals.

proposal has evidence in the discovery result, and *execute* only verified proposals with full provenance hashing. Each tool call carries explicit evidence—match mode, confidence score, and a cryptographic capability hash—making the selection an **evidence-carrying tool call** rather than a bare assertion. A deterministic verification function checks this evidence before execution; proposals lacking evidence are rejected even if they name real capabilities. The architecture enforces a **wall property**: no unverified selection reaches execution.

In the momentum example: if the LLM proposes `MOM_REV`, the gate rejects it—`MOM_REV` is a real strategy family, but "momentum" discovers only `MOM_WZS`. Likewise, `CARRY` and `VOL_TGT` exist but are not discovered from this prompt. The rejection is not based on validity (all are valid) but on evidence (only one was discovered).

This provides two properties. *Verification safety*: no unverified capability name reaches execution. *Replay determinism*: identical verified requests against identical data and registry produce bit-identical results, verifiable by an 8-layer hash that enables fault localization without re-execution.

Verification requires stable capability identity. The registry hashes each capability over its behavioral properties (semantic, algebraic, implementation), excluding discovery metadata by construction, so that adding aliases or descriptions never invalidates prior evidence or execution hashes (§3.2).

We evaluate on 30 prompts in 5 categories, with 4 strategy families and 9 metrics (36 valid family×metric pairs; discovery narrows to 1–3 per prompt). Against an assertion-only pipeline (schema validation, no verification), unwarranted executions occur at 23.3%; the verified pipeline reduces this to 10.0% (Fisher exact $p = 0.027$), eliminating them entirely on undiscoverable prompts (100%→0%). On adversarial prompts exploiting discovery-alias gaps, the verified pipeline performs worse—a tradeoff we discuss in §4.3. Repeated executions produce identical hashes. Verification adds under 14 ms.

Contributions.

1. **Evidence-carrying tool selection with a pre-execution verification wall.** Each tool call carries match evidence (mode, confidence, capability hash). A deterministic verification function checks this evidence before execution. The wall property guarantees that no unverified selection reaches the execution engine (§3.1).
2. **Fault-localizing execution hashes.** An 8-layer hash decomposes end-to-end provenance into distinct semantic layers, enabling diagnosis of what changed (data, registry, parameters, scoring) without re-execution (§3.3).
3. **Behavior-derived identity as the enabling mechanism.** Capability identity is computed from behavioral properties (semantic, algebraic, implementation), with discovery metadata excluded by construction. This makes verification evidence stable under registry evolution (§3.2).

2 Background

We consider systems where an LLM generates computational pipelines from natural-language prompts. The pipeline specifies a computational family (e.g., momentum, carry, mean-reversion, or volatility targeting), a

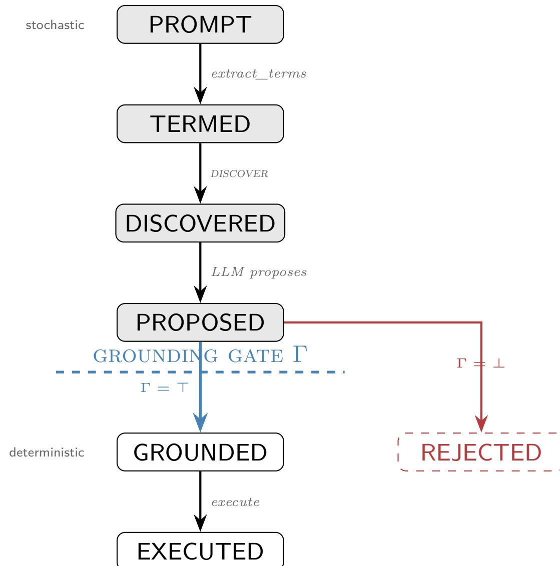

Figure 2: Pipeline state machine. The grounding gate Γ is the mandatory boundary between stochastic proposal and deterministic execution. Above the boundary, the LLM participates. Below, all functions are deterministic.

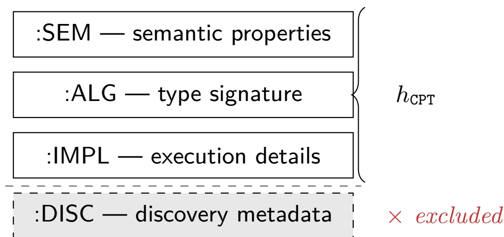

Figure 3: Capability hashing: three layers determine identity (h_{CPT}); the discovery layer is excluded. Changing aliases, descriptions, or tags does not change the capability hash.

scoring metric, parameter ranges, and a data source. Our evaluation domain is systematic trading research; the families serve as deterministic, semantically distinct computational pipelines for testing admissibility—the paper makes no claims about financial performance. The system expands these into a grid of concrete configurations (morphisms), executes each, scores the results, and selects the best.

The design space separates two concerns: *what to run* (the LLM's proposal) and *how to run it* (the deterministic execution engine). In existing systems, these concerns share a runtime with no formal boundary. An LLM may propose any name that appears in its training data, including names that are structurally valid but semantically unrelated to the user's request.

We define this failure mode precisely:

Definition 1 (Valid-but-unwarranted execution)**.** *An execution is* valid-but-unwarranted *if the proposed capability names pass schema validation (they exist in the system) but are not justified by the user's natural-language request.*

This is distinct from hallucination (proposing nonexistent names) and from malformed output (unparseable structure). Standard validation catches both of those. It does not catch valid-but-unwarranted proposals because the validator has no model of what the user intended—only of what the system supports.

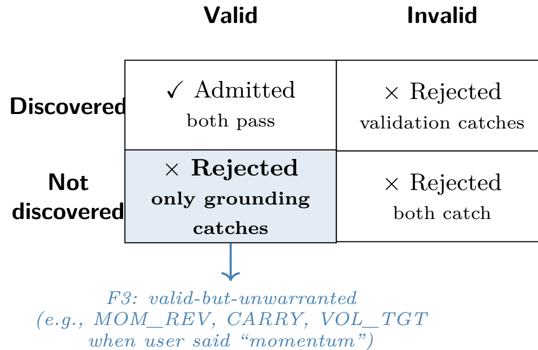

Figure 4: Grounding vs. validation. Schema validation partitions the family×metric space into valid (36 pairs) and invalid. The grounding gate further partitions valid pairs into discovered (1–3 per prompt) and not discovered. The bottom-left cell—valid but not discovered—is the valid-but-unwarranted region. Only grounding catches it.

Notation and terminology. Table 1 summarizes the key terms, capability names, and hash layers used throughout the paper. Strategy families and metrics are the evaluation domain's capability names; the architecture is domain-independent.

Table 1: Notation summary. Families and metrics are evaluation-domain identifiers; the architecture is domain-independent.

Symbol	Meaning	Context
Execution objects		
Request (FPR)	AI-proposed pipeline specification	Input to grounding
Morphism	Concrete configuration with all parameters bound	Grid expansion
Capability entry	Registry entry with layered hash	CPT registry
Execution footprint	Record of a completed execution	Output provenance
Evaluation-domain families (4)		
`MOM_WZS`	Momentum (z-scored)	Opposite of `MOM_REV`
`MOM_REV`	Mean-reversion	Opposite of `MOM_WZS`
`CARRY`	Carry	Overlaps `VOL_TGT` in NL
`VOL_TGT`	Volatility targeting	Overlaps `CARRY` in NL
Metrics (selected)		
`SRP`	Sharpe ratio	9 metrics total
`MDD`	Maximum drawdown	
Hash layers (8-layer execution hash)		
Registry	Hash of all capability entries	Sensitive to op changes
Request	Hash of proposed family, metric, parameters	Sensitive to intent
Morphisms	Hashes of all grid-expanded candidates	Sensitive to grid
Score / Selection	Hash of scoring result / best candidate	Sensitive to output
Data	Hash of source data	Sensitive to input

3 System Design

The system pipeline has three stages: *discover* capabilities from the user's terms, *ground* the AI-proposed request against the discovery result, and *execute* the grounded request with full provenance hashing. We describe each stage and the properties it provides.

3.1 Discovery and Grounding

Discovery. Given extracted terms from a natural-language prompt, AGENT-DISCOVER matches each term against the live capability registry. The registry contains 236 entries spanning operations (e.g., `dlog`, `wzs`), strategy families (`MOM_WZS`, `CARRY`, `MOM_REV`, `VOL_TGT`), and evaluation metrics (e.g., `SRP`, `MDD`). Matching uses a four-tier priority cascade:

1. *Exact*: the term equals a canonical name (confidence 1.0).
2. *Alias*: the term matches a registered alias (confidence 0.9). E.g., "sharpe" → `SRP`.
3. *Tag*: the term matches a semantic tag (confidence 0.7). E.g., "z-score" → `REL_MOM_Z`.
4. *Keyword*: the term appears as a substring in a name or description (confidence 0.5). E.g., "log returns" → `dlog`.

Terms with no match at any tier are placed in the UNRESOLVED set. The output is a discovery tuple containing a MATCHES list (each entry recording the term, the matched canonical name, the match tier, and the confidence) and an OK flag (true if at least one family and one metric were matched).

Discovery is deterministic: the same terms against the same registry produce the same result. The registry is queried live, not from a static allowlist—if a capability is added to the system, it becomes discoverable immediately without updating any configuration.

Grounding. The grounding function Γ takes a *request* (FPR—the AI-proposed pipeline specification, see Table 1) and the discovery result, and checks two conditions:

- The request's family field must appear as the canonical name of a FAMILY-kind match in the discovery result.
- The request's scoring metric field must appear as the canonical name of a METRIC-kind match in the discovery result.

If either condition fails, Γ returns ⊥ and execution cannot proceed. This is enforced architecturally: the runtime rejects any request that has not been grounded. A bare request without discovery evidence produces an error.

Property 1 (Grounding Safety)**.** *For all requests q and discovery results r: if $q.family \notin names(r)$ or $q.metric \notin names(r)$, then $\Gamma(q, r) = \bot$.*

The contrapositive: if Γ admits a request, every capability name in it was discovered from the user's terms.

Figure 2 shows the pipeline as a state machine. The critical boundary is between PROPOSED and GROUNDED. Above the boundary, the LLM participates—proposal is stochastic. Below the boundary, all functions are deterministic: morphism expansion, risk filtering, execution, scoring, and hashing depend only on the grounded request, the source data, and the registry state.

Why grounding is not validation. Schema validation checks whether a request's field values belong to the set of real capabilities. Grounding checks whether those values have evidence from the user's terms. In the motivating example, all four families—`MOM_WZS`, `MOM_REV`, `CARRY`, `VOL_TGT`—pass validation. But "momentum" discovers only `MOM_WZS`. Grounding rejects the other three, including `MOM_REV`: a real family built on the same infrastructure that produces the opposite computational signal. This distinction also separates grounding from constrained decoding, which restricts the LLM to the full set of 36 valid names but not to the per-prompt discovered subset (§5). Figure 4 illustrates this distinction.

3.2 Capability Identity

The capability registry assigns each entry a four-layer tuple. Three layers contribute to the capability's identity hash:

- **Semantic** (`:SEM`): algebraic properties—pure, deterministic, commutative, associative, idempotent, orientation, stateful.
- **Algebraic** (`:ALG`): type signature—input types, output types, arity, reversibility.
- **Implementation** (`:IMPL`): execution details—engine, IR variant, fusability, execution path.

The identity hash is computed as SHA-256 over the canonical serialization of these three layers:

$$h_{\mathrm{CPT}} = \mathrm{SHA256}(\textsc{sem} \,\|\, \textsc{alg} \,\|\, \textsc{impl})$$

A fourth layer, **Discovery** (`:DISC`), contains aliases, descriptions, tags, usage examples, and deprecation status. This layer is intentionally excluded from the identity hash (Figure 3).

The exclusion is the **description/identity separation** principle: a capability's identity is determined by what it *does* (semantic properties, algebraic structure, implementation), not by how it is *described* (natural-language metadata). Adding an alias like "log returns" → `dlog` changes what the LLM can discover; it does not change what `dlog` computes.

Why behavioral properties, not content. The identity hash h_{CPT} is derived from behavioral properties—purity, determinism, commutativity, type signature, arity, IR variant, fusability—rather than from content bytes or name strings. This distinguishes the design from content-addressed systems like Nix [8] and Unison [18], which hash source artifacts or ASTs. Hashing behavioral properties makes identity invariant to implementation refactoring that preserves semantics, and invariant to discovery-layer changes—which is what makes the verification evidence in §3.1 stable under registry evolution. A companion paper [12] proves that this identity hash is also necessary for safe compositional caching.

This has a practical consequence. A researcher who executed a pipeline six months ago can verify their execution hash against the current registry. If only descriptions, aliases, or tags were updated—a common maintenance activity—the capability hashes are unchanged, the registry hash is unchanged, and the execution hash validates. We verify this empirically: adding aliases (e.g., "logarithmic returns" → `dlog`), tags (e.g., `returns`, `log-space`), and extended descriptions to five operations used in the pipeline, rebuilding, and re-executing the same request produces identical hashes at every sub-hash layer (Table 5, DISC-layer row). Since `:DISC` is excluded from h_{CPT}, no change to discovery metadata propagates to h_C or to any downstream execution hash.

The full registry hash h_C is the SHA-256 of the sorted list of all capability hashes, producing a single fingerprint of the operation algebra. Every downstream hash includes h_C, making the entire provenance chain sensitive to changes in the capability set.

3.3 Execution and Provenance

Morphism expansion. A grounded request specifies a strategy family, scoring metric, and parameter ranges. The system expands parameter ranges into a grid via Cartesian product. Each grid point produces a *morphism*: a concrete configuration with all parameters bound (e.g., a specific signal window width, risk window, and leverage cap). Each morphism is canonicalized and hashed. The hash includes the canonical expression, bound parameters, and the registry hash h_C.

Risk filter. Each morphism is evaluated against four admissibility predicates: (1) all operations are resolved and plannable, (2) the source data file exists, (3) all parameters are non-negative, and (4) estimated leverage is within bounds. A morphism is *admissible* if all four predicates hold. Inadmissible morphisms are classified as rejected with the failing predicates listed.

Execution. Each admissible morphism is executed through a typed compilation pipeline (parse → normalize → canonicalize → plan → optimize → execute). Execution produces an *execution footprint* recording the bound parameters, the computation graph (DAG of operations), and the output artifacts with content hashes.

Table 2: Execution hash decomposition. Each sub-hash covers a distinct semantic layer, enabling fault localization without re-execution.

Layer	Sub-hash	Covers
Registry	`DIC_HSH`	Operation algebra (all capability hashes)
Request	`FPR_HSH`	Proposed family, metric, and parameters
Morphisms	`MOR_HSHS`	All grid-expanded candidate expressions
Plans	`SRH_HSHS`	Compiled execution plans
Artifacts	`CMPN_HSHS`	Output data artifacts
Score	`SCR_HSH`	Scoring result (metric values)
Selection	`SEL_HSH`	Best candidate identity
Data	`DATA_HSH`	Source data content

Execution hash. The end-to-end execution hash decomposes into eight sub-hashes (Table 2):

The execution hash is the SHA-256 of all eight sub-hashes concatenated. This provides **replay determinism**: identical inputs and registry state produce identical hashes. When two hashes differ, comparing sub-hashes localizes the divergence. A change in the registry hash indicates the operation algebra was modified. A change in the data hash indicates the source data changed. A change in the score hash with everything else equal indicates the scoring computation changed. This fault-localization property allows diagnosis without re-execution.

4 Evaluation

We evaluate four claims: (1) the gate is exact—zero false admits, zero false rejects on known grounding status, (2) execution hashes are deterministic across repeated runs and detect targeted changes, (3) an assertion-only pipeline silently executes unwarranted capabilities that the verified pipeline rejects, and (4) the verification pipeline adds negligible overhead.

4.1 Grounding Gate Selectivity

Setup. We construct 96 request tuples across 6 discovery contexts spanning all 4 families: 27 with both family and metric present in the discovery result, and 69 with at least one field absent. The 69 ungrounded cases include: cross-family rejection (e.g., `CARRY` when the discovery terms are "momentum" and "sharpe"), semantic inversion (`MOM_REV` when only "momentum" is discovered), plausible but nonexistent names (`TREND_FOLLOW`, `TRACKING_ERR`), multi-family contexts where undiscovered families are rejected, and fabricated names.

Results. Table 3 reports the confusion matrix. The grounding gate admits all 27 grounded proposals and rejects all 69 ungrounded ones. **Zero false admits. Zero false rejects.**

Table 3: Grounding gate confusion matrix (96 tuples, 4 families, 6 discovery contexts). The gate is exact: it admits exactly the proposals whose names appear in the discovery result.

	Admitted	Rejected
Grounded (27)	27	0
Ungrounded (69)	0	69

The gate is exact: it admits exactly the proposals whose names appear in the discovery result. It is neither conservative (rejecting valid proposals) nor permissive (admitting ungrounded ones).

4.2 Replay Determinism and Drift Detection

Replay. We execute five request configurations 10 times each, recording the execution hash and all eight sub-hashes. Table 4 reports the results: all 50 runs produce identical top-level execution hashes within

each configuration, implying identical sub-hash decompositions. Determinism holds across CSV parsing, multi-column frame operations, canonical serialization, and parameter grid expansion.

Table 4: Replay determinism across 5 configurations × 10 repetitions. Every sub-hash matches within configuration. Configs span all 4 families and 3 metrics.

Family / Metric	Parameters	Reps	Unique	Match
Momentum / Sharpe	window=25	10	1	✓
Carry / Sharpe	(defaults)	10	1	✓
Mean-reversion / Sharpe	window=25	10	1	✓
Vol-targeting / Volatility	window=60	10	1	✓
Momentum / Max-drawdown	window=50	10	1	✓

Drift detection. We introduce targeted changes and re-execute the same request (Table 5). All configurations use a 3-morphism sweep (signal window $\in \{25, 50, 100\}$) so that metric and data changes can affect selection. For data drift, we replace the file contents at a *stable path* so the request hash is unchanged—isolating the effect of data from the effect of the request.

Table 5: Drift detection: which sub-hashes change under targeted modifications. "=" means unchanged; "Δ" means changed. The data-drift row uses a stable source path so the request hash is unchanged. The DISC-layer row validates description/identity separation: adding aliases, tags, and descriptions to five pipeline operations produces identical hashes at every layer.

Change type	Registry	Request	Morphisms	Plans	Score	Selection
Scoring metric	=	Δ	=	=	Δ	Δ*
Parameter value	=	Δ	Δ	Δ	Δ	Δ
Source data	=	=	=	=	Δ	Δ
DISC-layer metadata	=	=	=	=	=	=

*Selection hash changes when the best morphism differs under the new metric; with a single-morphism grid it would not.

4.3 Assertion-Only Pipeline Comparison

Setup. We compare two pipelines on 30 prompts in five categories (6 straightforward, 8 confusable, 8 adversarial, 4 undiscoverable, 4 multi-family). Pipeline A is the *verified* pipeline: discovery → LLM (sees discovered names only) → grounding gate → execution. Pipeline B is the *assertion-only* pipeline: the LLM receives all 36 valid family×metric pairs, proposes a pipeline, validates against a JSON schema with enum-constrained fields, and executes directly—the selection is asserted but never verified against the user's terms.

The four strategy families—`MOM_WZS` (momentum), `MOM_REV` (mean-reversion), `CARRY` (carry), `VOL_TGT` (volatility targeting)—are semantically distinct but pairwise confusable. `MOM_WZS` and `MOM_REV` share identical infrastructure with opposite computational signals. `CARRY` and `VOL_TGT` share overlapping natural-language descriptions. Both pipelines use Claude Sonnet at temperature 0.7 with three repetitions per prompt.

Failure taxonomy. We classify each execution using a failure taxonomy (Table 6):

Results. Table 7 reports failure rates by category.

The verified pipeline reduces unwarranted executions from **23.3%** to **10.0%** overall (Fisher exact test, two-sided $p = 0.027$). The gate eliminates them entirely on undiscoverable prompts (0/12 vs. 12/12). Both pipelines achieve 0% on straightforward and multi-family prompts. The residual 10.0% in Pipeline A (9/90)

Table 6: Failure taxonomy. F3–F5 are the failure modes that only the grounding gate catches; schema validation admits all three because the proposed names are real.

Code	Failure mode	Validation	Grounding	Risk
F0	Correct	—	—	—
F1	Malformed output	catches	catches	—
F2	Hallucinated name	catches	catches	—
F3	**Valid-but-unwarranted**	**misses**	**catches**	—
F4	**Metric swap**	**misses**	**catches**	—
F5	**Partially grounded**	**misses**	**catches**	—
F6	Parameter nonsense	partial	misses	catches

Table 7: Unwarranted execution (F3) rates by category and pipeline (90 trials each, Claude Sonnet, $T = 0.7$, 3 reps). F3 includes semantic inversions (e.g., proposing mean-reversion when momentum was intended). Classification is intent-based: the proposed capability does not match the human-annotated expected capability.

Category	Pipeline	N	F3	F3 rate
Straightforward (6)	A (verified)	18	0	0%
	B (assertion-only)	18	0	0%
Confusable (8)	A (verified)	24	4	16.7%
	B (assertion-only)	24	6	25.0%
Adversarial (8)	A (verified)	24	5	20.8%
	B (assertion-only)	24	3	12.5%
Undiscoverable (4)	A (verified)	12	0	0%
	B (assertion-only)	12	12	100%
Multi-family (4)	A (verified)	12	0	0%
	B (assertion-only)	12	0	0%
Overall	**A (verified)**	**90**	**9**	**10.0%**
	B (assertion-only)	**90**	**21**	**23.3%**

arises from confusable and adversarial prompts where discovery returns multiple families and the LLM picks the wrong one—a within-discovery-set error that verification alone cannot prevent.

The undiscoverable category is the cleanest contrast: prompts like "trend-following strategy" discover no family ("trend" is not an alias for any registered family). The verified pipeline correctly rejects execution. The assertion-only pipeline picks a family from the full enum and executes silently (100% F3).

The adversarial tradeoff. On adversarial prompts, the verified pipeline has a *higher* F3 rate (20.8%) than the assertion-only pipeline (12.5%). The strongest case is prompt **A02**: "momentum strategy—but bet against recent winners." The user intends mean-reversion (`MOM_REV`), but "momentum" discovers only `MOM_WZS`. The verified pipeline, constrained to the discovered set, proposes `MOM_WZS`—the opposite computational signal. The assertion-only pipeline, seeing all valid names, correctly proposes `MOM_REV` in all 3 repetitions. The grounding gate prevents unverified capabilities from reaching execution, but it cannot prevent wrong selection when the correct capability falls outside the discovery system's alias coverage.

Conversely, prompt **C02** ("momentum strategy that captures reversals") shows the gate's value on confusable prompts: the assertion-only pipeline proposes `MOM_REV` (the opposite signal) in all 3 repetitions, while the verified pipeline, constrained to the discovered set ("momentum" $\rightarrow$ `MOM_WZS` only), proposes the correct family. Schema validation cannot distinguish them—both are valid families, both produce well-formed output.

4.4 Overhead

Table 8 reports pipeline overhead measured over 20 repetitions. Discovery and grounding together add under 14 ms per request, dominated by process startup rather than computation.

Table 8: Pipeline overhead (20 repetitions). Discovery and grounding add under 14 ms per request.

Stage	Time (ms)
Agent-Discover	13.6 ± 0.9
Agent-Ground	13.2 ± 0.6
Agent-Dryrun (incl. discovery + grounding)	13.4 ± 0.8

The cost of the grounding gate is negligible relative to both LLM latency (typically 1–3 seconds) and full pipeline execution.

F3 residual. The verified pipeline's 10% uncaught error rate (Table 7) consists entirely of F3 (valid-but-unwarranted) and F3_INV (semantic inversion) failures—cases where the proposed capability name exists in the discovery set but does not match the user's intent. The gate catches proposals referencing undiscovered capabilities (F1, F2) but cannot distinguish correct from incorrect selection within the discovered set. Reducing F3 requires richer discovery semantics (e.g., intent-aware matching), not a stronger gate.

5 Related Work

Tool-augmented LLMs. ReAct [1], Toolformer [2], and frameworks like LangChain [3] enable LLMs to call external tools. In all these systems, tool selection is an assertion: the LLM selects which tool to invoke with no verification boundary between selection and execution. Gorilla [11] improves selection accuracy by retrieving API documentation, but the LLM selects freely from retrieved candidates—retrieval constrains context, not execution. A valid but wrong tool selection produces a silent failure. Our grounding gate converts selection from assertion to verified claim by requiring evidence linking each selection to the user's terms.

Structured output and constrained decoding. OpenAI structured outputs [4] and Outlines [5] constrain LLM output to conform to a JSON schema or grammar. With enum fields, constrained decoding eliminates hallucinated names (F2). However, the enum encodes the full set of valid values—all 36 family×metric pairs in our evaluation—not the per-query discovered subset (typically 1–3 pairs). Constrained decoding prevents hallucinated names but not unwarranted ones: the LLM may select `MOM_REV` for a momentum prompt because it is a valid enum value. The grounding gate restricts to the discovered subset (Table 6).

Tool contracts and metadata. ToolGate [13] proposes Hoare-style pre/post-condition contracts for LLM tool calls, verifying that tool inputs satisfy declared preconditions. This addresses input validation (are the arguments well-formed?) but not selection validation (is this the right tool?). Spring AI [14] attaches structured metadata (return types, tags) to tool definitions, improving LLM selection through richer context. Neither system provides the evidence-carrying property we require: a cryptographic link between the selection decision and the user's discovery terms, verified before execution.

Experiment tracking. MLflow [6] and Weights & Biases [7] record execution artifacts and enable comparison across runs. AER [16] defines a structured schema for logging LLM reasoning traces. These provide post-hoc observability: researchers can see what happened. They do not provide pre-execution verification: they cannot prevent an unwarranted execution from occurring. Our system provides both—the grounding gate prevents unwarranted execution, and the 8-layer execution hash provides post-hoc traceability with fault localization.

Content-addressed computation and identity. Nix [8] and Guix [9] content-address software builds, ensuring reproducibility through cryptographic hashing of all build inputs. Software Heritage [17] assigns persistent identifiers (SWHIDs) to source code artifacts. AGNTCY Agent Discovery Service [15] content-addresses AI agent descriptions on a distributed hash table, enabling decentralized agent discovery. Our execution hash applies content-addressing to research pipelines. The key difference is in *what* is hashed: Nix hashes all package metadata (descriptions, maintainer fields, etc.) into the derivation hash; Software Heritage and AGNTCY hash source artifacts or description content. Our behavior-derived identity hashes *behavioral properties* (semantic, algebraic, implementation), not source content or descriptions, making identity invariant to description changes and implementation refactoring that preserves semantics.

Identity models. Unison [18] hashes functions over ASTs, making identity invariant to renaming. The Semantic Web identity literature [21, 22, 23] establishes the principle that a resource's identity should be independent of its descriptions. Zooko's triangle [24] frames the tension between human-meaningful, decentralized, and secure names. Our identity model instantiates these principles as an enabling mechanism for verified selection: behavioral hashes provide secure identity, while human-meaningful aliases live in the excluded discovery layer.

Proof-carrying code. Necula's proof-carrying code (PCC) [19] attaches a machine-checkable proof of memory safety to compiled binaries; the consumer verifies the proof before execution without re-analyzing the source. Our evidence-carrying tool selection is a structural analog: each tool call carries evidence (match mode, confidence, capability hash) that is verified before execution. The analogy is structural, not formal—PCC proves memory safety via type-theoretic certificates; we verify selection justification via discovery evidence. Both enforce the same architectural pattern: proof travels with artifact, is verified before execution, and verification is cheaper than re-derivation. Justification Logic [20] formalizes this pattern as $t : \phi$ ("t is evidence for ϕ"), providing a theoretical framework for evidence-carrying assertions.

Typed APIs and DSLs. Typed APIs and domain-specific languages constrain the space of valid programs. Languages like Stan [10] for probabilistic programming provide type safety and domain-specific validation. These systems prevent structurally invalid programs but do not address the prompt-to-program gap: they assume the programmer (or LLM) already knows the correct program to write. Our contribution is the discovery-then-verification pipeline that constrains which programs the LLM *may* propose, based on evidence from the user's natural-language terms.

Positioning. Table 9 summarizes the property coverage across systems. No existing system we are aware of provides evidence-carrying tool selection with a pre-execution verification wall. Content-addressed systems provide replay determinism; tool-contract systems provide input validation; experiment trackers provide post-hoc observability. Our contribution is the verification boundary itself: tool selection must carry evidence, and a deterministic function checks that evidence before execution. The identity model (§3.2) is the enabling mechanism that keeps this evidence stable under registry evolution.

6 Discussion and Limitations

Grounding is syntactic, not semantic. The grounding gate checks name membership: is the proposed family name in the discovery result? It does not check whether the user's research question is well-posed, whether the chosen family is the *best* match, or whether the parameter values make scientific sense. An LLM can still make bad research decisions within the grounded vocabulary. The gate prevents the LLM from selecting capabilities it was not offered; it does not prevent poor selection among offered capabilities.

Discovery coverage determines the gate's ceiling. The grounding gate can only admit capabilities that the discovery system finds. When a user's terms describe a capability indirectly—using infrastructure terminology rather than family names—discovery may fail to match, and the gate will block a correct proposal or force a wrong selection from the discovered set. Our evaluation confirms this: on adversarial prompts that exploit discovery-alias gaps, the verified pipeline has a higher unwarranted-execution rate than the

Table 9: Property coverage across systems. • = full, ○ = partial, — = absent.

	Verif. gate	Evidence-carrying	Replay determ.	Fault localiz.	Behav.-derived id.	Desc./id. sep.	Live registry
Tool-aug. LLMs	—	—	—	—	—	—	○
Constrained decoding	○	—	—	—	—	—	—
ToolGate [13]	○	—	—	—	—	—	—
Spring AI [14]	—	—	—	—	—	—	•
AGNTCY [15]	—	—	—	—	—	—	•
Experiment tracking	—	—	○	○	—	—	—
Nix/Guix	—	—	•	○	—	—	—
Unison [18]	—	—	•	—	○	—	—
Software Heritage	—	—	•	—	—	—	—
Typed APIs/DSLs	○	—	—	—	—	—	—
This work	•	•	•	•	•	•	•

assertion-only pipeline (§4.3). Improving alias coverage in the discovery layer is the primary path to reducing this limitation, and it can be done without changing the grounding mechanism itself.

Single domain. We evaluate in systematic trading research. The architecture's components—discovery, grounding, layered hashing—are domain-independent, but we have not demonstrated them in other domains. Generalizing to, e.g., molecular simulation or climate modeling pipelines would require building domain-specific capability registries. The grounding mechanism itself—match terms, check membership—is domain-independent.

Capability vocabulary. The evaluation registry contains 4 strategy families and 9 metrics (36 valid combinations). The architecture imposes no limit on capability count; the registry currently contains 236 entries spanning operations, families, metrics, signal blocks, and recipes. Discovery and grounding operate identically regardless of registry size. Evaluating with larger vocabularies—including vocabularies where discovery ambiguity increases—is future work.

Conservative registry fingerprint. The registry hash fingerprints the entire operation algebra. Adding an unrelated operation changes the registry hash, which cascades to the execution hash, even though the new operation has no effect on the pipeline. Per-operation dependency tracking (a finer-grained hash that includes only the operations actually used) is future work that would reduce false-positive drift alerts.

Connection to compositional caching. The identity hash constructed in §3.2 has consequences beyond verification. A companion paper [12] proves that the same behavioral hash that makes the gate's evidence stable also makes compositional cache keys safe—the grounding gate and the caching system share the same identity foundation.

The evidence-carrying pattern. The grounding gate instantiates a pattern broader than tool selection: *evidence travels with the artifact and is verified before use.* This is structurally analogous to proof-carrying code [19], where a compiled binary carries a proof of memory safety that the consumer verifies before loading. In our system, each tool call carries discovery evidence (match mode, confidence, capability hash) that the grounding function verifies before execution. The analogy is structural, not formal: PCC proves type safety; we verify selection justification. But the architectural invariant is the same—the consumer never trusts the producer's assertion without checking the attached evidence. This pattern may generalize to other AI-driven systems where stochastic reasoning produces artifacts consumed by deterministic infrastructure.

7 Conclusion

Tool selection in AI-driven research is an assertion that can be wrong. The grounding gate converts it to a verified claim: each selection must carry evidence linking it to the user's terms, and a deterministic verification function checks this evidence before execution. On undiscoverable prompts, the gate eliminates unwarranted executions entirely (100%→0%); on adversarial prompts that exploit discovery-alias gaps, it forces wrong selections—a tradeoff inherent to constraining the admissible set rather than the valid set.

Eight-layer execution hashing decomposes end-to-end provenance into distinct semantic layers, providing deterministic replay across all 50 runs in our evaluation and enabling fault localization without re-execution. The enabling mechanism—behavior-derived identity with discovery metadata excluded by construction—keeps verification evidence stable as the registry evolves; we verify empirically that enriching five pipeline operations produces identical hashes at all layers. A companion paper [12] proves that this same identity hash is necessary for safe compositional caching downstream of the gate.

The grounding gate addresses the assertion gap: the distance between a tool selection that is structurally valid and one that is verified against the user's intent. Schema validation, constrained decoding, and typed APIs prevent malformed or nonexistent proposals; the grounding gate addresses the unwarranted ones that those mechanisms miss. The overhead is under 14 ms.

References

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

[2] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023.

[3] H. Chase. LangChain, 2023. `https://github.com/langchain-ai/langchain`.

[4] OpenAI. Structured outputs, 2024.

[5] B. T. Willard and R. Louf. Efficient guided generation for large language models. *arXiv:2307.09702*, 2023.

[6] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zuber. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41(4), 2018.

[7] L. Biewald. Experiment tracking with Weights and Biases, 2020. `https://www.wandb.com`.

[8] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, 2004.

[9] L. Courtès. Functional package management with Guix. In *European Lisp Symposium*, 2013.

[10] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017.

[11] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive APIs. *arXiv:2305.15334*, 2023.

[12] T. Dionysopoulos. Canonical execution semantics for stochastic program generators. Companion paper, 2026.

[13] Q. Tang, Z. Deng, H. Lin, H. Han, Q. Liang, and B. Liu. ToolGen: Unified tool retrieval and calling via generation. *arXiv:2410.03439*, 2024.

[14] Spring AI. Structured tool metadata for model function calling, 2025. `https://docs.spring.io/spring-ai/reference/`.

[15] AGNTCY. Agent Discovery Service: Content-addressed agent descriptions on DHT, 2025. `https://github.com/agntcy/`.

[16] Agent Evidence Record (AER). Structured schema for LLM reasoning trace logging, 2025.

[17] R. Di Cosmo and S. Zacchiroli. Software Heritage: Why and how to preserve software source code. In *iPRES*, 2017.

[18] P. Chiusano and R. Bjärnason. Unison: A friendly programming language from the future, 2015. `https://www.unison-lang.org/`.

[19] G. C. Necula. Proof-carrying code. In *POPL*, 1997.

[20] S. N. Artemov. The logic of justification. *Review of Symbolic Logic*, 1(4):477–513, 2008.

[21] W3C Technical Architecture Group. httpRange-14: What is the range of the HTTP dereference function?, 2005.

[22] I. Jacobs and N. Walsh. Architecture of the World Wide Web, Volume One. W3C Recommendation, 2004.

[23] A. Miles and S. Bechhofer. SKOS Simple Knowledge Organization System Reference. W3C Recommendation, 2009.

[24] Z. Wilcox-O'Hearn. Names: Distributed, secure, human-readable: Choose two, 2001.

A Prompt Dataset

The comparison experiment (§4.3) uses 30 prompts organized into five categories:

- **Straightforward** (6): Clear term-to-capability mapping. E.g., "Momentum strategy on At.csv ranked by Sharpe." (discovers `MOM_WZS` + `SRP`.)
- **Confusable** (8): Terms that overlap between families. E.g., "Momentum strategy that captures reversals." ("momentum" discovers `MOM_WZS`; "reversals" is ambiguous between `MOM_WZS` and `MOM_REV`.)
- **Adversarial** (8): Language designed to elicit valid-but-wrong proposals. E.g., "Momentum strategy—but bet against recent winners." (discovers `MOM_WZS` only; user intends `MOM_REV`.)
- **Undiscoverable** (4): Terms with no family match. E.g., "Trend-following strategy ranked by Sharpe." ("trend" discovers no family.)
- **Multi-family** (4): Prompts explicitly naming two families. E.g., "Compare momentum and carry on At.csv." (discovers `MOM_WZS` + `CARRY`.)

Each prompt is annotated with expected family, expected metric, discovery terms, and expected confusion type. The complete dataset is in `experiments/prompts_30.json`.

B Failure Taxonomy Details

F3, F4, and F5 are the critical failure modes that distinguish grounding from validation. In all three cases, the proposed capability names are real and pass schema validation. The grounding gate catches proposals whose names lack evidence from the user's discovery terms—eliminating F3 when the correct capability was never discovered (e.g., undiscoverable prompts: 100%→0%). When the correct and incorrect capabilities are both discovered (e.g., adversarial prompts where alias coverage is incomplete), the gate admits both; wrong selection within the discovered set is a residual failure the gate does not address (§4.3).

Table 10: Full failure taxonomy with detection mechanisms.

Code	Mode	Stage	Valid.	Ground.	Risk
F0	Correct	Completes	—	—	—
F1	Malformed	Parse	✓	✓	—
F2	Hallucinated name	Validation	✓	✓	—
F3	Valid-but-unwarranted	Grounding	×	✓	—
F4	Metric swap	Grounding	×	✓	—
F5	Partially grounded	Grounding	×	✓	—
F6	Parameter nonsense	Risk filter	◦	×	✓
F7	Registry drift	Replay	×	×	×
F8	Provenance gap	Audit	×	×	×

C Measured Overhead

Overhead measurements from 20 repetitions on the evaluation hardware:

- Agent-Discover: 13.6 ± 0.9 ms (median 13.0 ms, range 13–16 ms)
- Agent-Ground: 13.2 ± 0.6 ms (median 13.0 ms, range 13–15 ms)
- Agent-Dryrun (includes internal discovery + grounding): 13.4 ± 0.8 ms (median 13.0 ms, range 13–16 ms)

All three operations are dominated by process startup overhead (the BLISP subprocess). The computation itself is sub-millisecond. Discovery and grounding together add under 14 ms per request—negligible relative to LLM latency (typically 1–3 s).

D Artifact Repository

Experiment scripts, expected outputs, prompts, and verification tools are in the `artifacts/grounding-gate/` directory. Registry snapshots (capability dictionary, alias tables, family definitions) are in `artifacts/shared/registry/`. To verify all paper claims against existing results without re-running experiments:

```
bash reproducibility/verify_all.sh
```

Deterministic experiments (selectivity, replay, drift, overhead) require only the `blisp` binary and no API key. The comparison experiment (Table 7) requires an LLM API key and makes 180 API calls.