Behavioral Hot Paths: Discrete, Cost-Aware Discovery of Recurring Agent Behavior

Shane Rohan Barakat[1]
shane@polarity.so
[1]Research Division, Polarity Labs

June 11, 2026

Abstract

Language agents repeat themselves: a coding agent clones, builds, and tests; an analytics agent connects, queries, and formats. Each repetition is billed at full frontier cost even after the agent has performed the behavior thousands of times. Compiling a recurring behavior into a cheap specialist is a natural response, but it requires first answering a question prior work leaves implicit: how does an agent *discover*, unsupervised, which of its behaviors recur often enough to be worth replacing? We argue this discovery step must be *discrete* rather than embedding-based, since vectorizing behaviors and clustering by distance makes the central quantity—how repetitive a workload is—a function of an encoder and a threshold rather than a fact about the agent. We instead reduce each typed operation to a canonical *signature* under an inspectable normalization, treat a behavior as a frequent subsequence over the resulting alphabet (mined by classical frequent-pattern algorithms), and decide whether to specialize by an *economic* criterion: compile only when projected savings on the model-generated step, net of the specialist's accuracy and the cost of verifying its output and falling back when it errs, exceed amortized training cost. Training data is harvested along discovered behaviors and screened by an automatic verifier, with an LLM judge confined to screening concrete outputs rather than judging similarity. On 57,939 real agent-execution trace events we validate the method's premise and discovery machinery: typed operations collapse 46× with no encoder and the top nine types cover 94.6% of calls; frequent-subsequence mining recovers the canonical work loop; and the model-generated step shows a real cost spread (median $0.025, mean $0.22) against a cheap-model tier 40–60× lower. We are deliberately explicit about what the traces do *not* yet show—a per-behavior break-even, a harvest-yield figure, and a trained specialist matching the frontier—and give the protocol for those measurements. We present this as the discovery foundation for self-specializing agents, scoped to behaviors that admit an automatic correctness oracle.

1 Introduction

A language agent executing a workload tends to walk the same routes repeatedly. The surface requests differ ("revenue by region last quarter," "cancellations by month"), yet the underlying behavior is identical: connect to a database, generate an aggregation query, format the result. Today every such execution is served by a frontier model at full cost, even when the agent has demonstrably performed the behavior thousands of times. If a behavior recurs and the step that costs money is one the model *generates*, that step is a candidate to be compiled into a small specialist that produces the same output far more cheaply, with the frontier retained as a fallback.

The compilation itself, distilling a specialist and gating its deployment, is comparatively well understood. The unanswered question is the one that comes first: *how does the agent discover, with no human labeling, which behaviors recur often enough that compiling them pays off?* This paper is about that discovery step.

Our central methodological claim is that discovery must be **discrete**. The tempting approach is to embed each behavior as a vector and cluster by cosine distance, declaring two behaviors "the same" when they fall within a threshold. We argue this is the wrong primitive for a system whose headline quantity is the degree of repetition in a workload. Under vectorization, that quantity is determined by an arbitrary encoder and an arbitrary distance cutoff: change either and the answer changes, so "80% of work falls into ten behaviors" becomes a statement about the encoder rather than about the agent. Worse, embedding distance has no grounding in execution; two queries differing only in a literal are functionally identical but may embed far apart, while superficially similar calls may do entirely different work.

The alternative we develop keeps every decision discrete and inspectable. Agent operations are *typed*: a tool call is a function name with structured arguments. We reduce each operation to a canonical *signature* that retains its structure and abstracts away volatile arguments, so that operations which should count as identical become *literally equal* rather than merely close. A behavior is then a frequent subsequence over the signature alphabet, recovered by classical frequent-pattern mining whose only parameter is a

frequency count with a clear meaning. Whether to compile a discovered behavior is decided not by any similarity cutoff but by an *economic* break-even: projected savings on the behavior's model-generated step against the one-time cost of training the specialist. Finally, training data harvested along a discovered behavior is screened by an automatic verifier; where no execution oracle exists, an LLM judge may screen the concrete extracted outputs, but the judge never decides what counts as the same behavior.

This paper contributes: (i) a discrete formulation of agent-behavior discovery via canonical signatures and frequent-subsequence mining, with an explicit account of why this is preferable to embedding-based clustering; (ii) an economic specialization criterion that replaces an arbitrary similarity threshold with a workload-grounded break-even on model-generated steps, accounting for specialist accuracy and the cost of online verification and frontier fallback; (iii) a verified-harvesting procedure that turns discovered behaviors into clean training data while confining LLM-judge usage to a defensible role; and (iv) an empirical validation on real production traces (concentration along the canonicalization ladder, recovery of frequent behavioral subsequences, and the real cost distribution of the model-generated step), together with an explicit account of the measurements that remain (a per-behavior break-even, harvest yield, and a trained specialist) and the protocol for obtaining them. We scope the method to behaviors that admit an automatic correctness oracle and discuss the boundary explicitly.

2 Related Work

Hot-path discovery and dynamic compilation. Trace-based just-in-time compilers profile a running program, identify frequently executed paths, and compile those paths into optimized code while leaving the cold remainder interpreted [1]. Our system performs the analogous move at the level of agent behavior: profile the trace, find hot paths, and compile them into cheap specialists, with the frontier model as the interpreter of last resort. The unit being canonicalized is a typed operation rather than a basic block, and canonicalization draws on standard normalization ideas from compilers [2] and, for the database setting, semantic query equivalence [3].

Frequent-pattern and episode mining. Recovering recurring behavior from a corpus of traces is an instance of sequential-pattern mining [4, 5] and frequent-episode discovery in event sequences [6]. These provide exact, support-based algorithms over a discrete alphabet, which is precisely what we need to avoid embedding-distance thresholds.

Agents and tool use. Prompting and acting frameworks let a frozen model plan and call tools [7, 8], and recent systems treat agent pipelines as programs to be compiled and optimized [9]. We complement this line by mining the resulting execution traces for repeated structure worth specializing.

Distillation, specialization, and routing. Knowledge distillation transfers teacher behavior to a smaller student [10], and LoRA makes per-behavior specialization cheap to train and serve [11]. Routing and cascades trade cost for quality across a pool of models [12]; semantic caching reuses answers for near-duplicate inputs [13]. Our contribution is upstream of all of these: deciding *which* behaviors to build specialists (or caches) for, and supplying their training data.

Self-generated supervision and verification. A model can be improved from its own outputs [14, 15, 16], but imitating unverified outputs is fragile, motivating verification. Training verifiers to check outputs is a standard route to reliability [17], with executable unit tests serving as the oracle for code [18] and execution against a reference database for SQL [20, 21]. LLM-as-judge is widely used for open-ended evaluation [19]; we deliberately restrict it to screening concrete harvested outputs, never to deciding behavioral identity, because judging similarity reintroduces exactly the non-reproducibility we set out to avoid.

3 Problem Setup

An agent executing a task emits a *trace*, a finite sequence of typed operations

$$\rho = \langle o_1, o_2, \ldots, o_L \rangle, \qquad o_i = (\text{name}_i,\ \text{args}_i),$$

where name_i is a tool or action identifier and args_i its structured arguments. A corpus $R = \{\rho^{(1)}, \ldots, \rho^{(M)}\}$ collects M such traces from a workload. We assume operations are observable at this granularity (the deployment records tool calls with their arguments, not merely that a tool was used); the empirical study of §6 draws on a deployment that captures exactly this. The goal of discovery is to identify, from R alone and without supervision, the recurring behaviors whose compilation into specialists is economically justified, and to produce verified training data for them.

4 Method

The pipeline has four discrete stages (Figure 1): reduce operations to canonical signatures (§4.1); mine frequent behaviors over the signature alphabet (§4.2); decide which behaviors to compile by an economic criterion (§4.3); and harvest verified training data along the chosen behaviors (§4.4). No stage uses a similarity threshold.

4.1 Canonical Signatures

A *canonicalization* map κ sends an operation to a *signature* $\sigma = \kappa(o)$ that retains the operation's structure and abstracts its volatile arguments. For a database query, κ keeps the normalized query skeleton (its abstract syntax tree with literals, aliases, and whitespace removed) and discards the specific constants; for an HTTP call, it keeps the endpoint template and discards path parameters; for a repository clone, it keeps the operation and discards the URL. Two operations are deemed equivalent exactly when their signatures are equal,

$$o_i \equiv_\kappa o_j \iff \kappa(o_i) = \kappa(o_j),$$

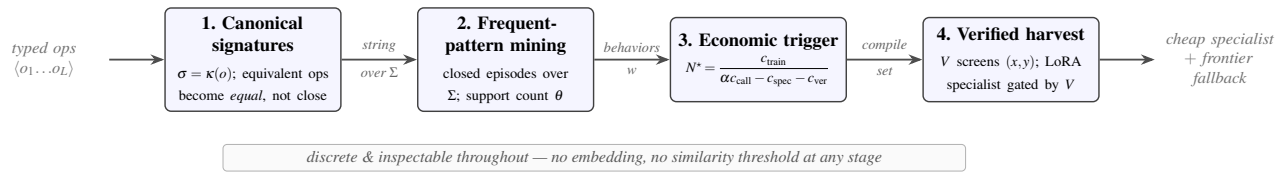

Figure 1: The discovery pipeline. Four discrete stages turn a corpus of typed operation traces into deployable specialists: canonical signatures (§4.1) make equivalent operations *literally equal*; frequent-pattern mining (§4.2) recovers recurring behaviors over the signature alphabet; an economic break-even (§4.3) decides which to compile (and separates caching from specialization); and verified harvesting (§4.4) produces clean training data, with the same verifier V later gating deployment. No stage uses a similarity threshold.

a discrete test anyone can read off the normalization rules, in contrast to thresholding a distance. The set of signatures observed in R forms a finite alphabet Σ.

Two properties of κ matter downstream. First, granularity is a modeling choice, and we treat it as a *ladder* of increasingly abstract maps

$$\kappa_0 \preceq \kappa_1 \preceq \kappa_2 \preceq \kappa_3,$$

from exact byte equality (κ_0), through literal-normalized and structure-normalized forms (κ_1, κ_2), to execution or semantic equivalence (κ_3, e.g. two SQL skeletons that provably return the same result [3]). More abstract maps induce more collisions and therefore higher apparent repetition; rather than hide this, we report discovery as a function of the ladder (§6, §7). Second, we attach to each signature a predicate $g(\sigma) \in \{0,1\}$ indicating whether its variable slot is *model-generated* (the SQL the agent writes, the field it extracts) as opposed to deterministic plumbing (a fixed clone or fetch). Only model-generated steps are candidates for a learned specialist. A deterministic step is not a model cost at all; its recurrence may justify ordinary caching of an expensive external call, an optimization orthogonal to the frontier-replacement question we study here. In a real deployment g is read off the event type that produced the step (§6); we flag where the trace schema makes this annotation clean and where it does not.

4.2 Behavior Discovery by Frequent-Pattern Mining

Under κ, each trace becomes a string over Σ,

$$s(\rho) = \kappa(o_1)\,\kappa(o_2)\cdots\kappa(o_L) \;\in\; \Sigma^*.$$

A *behavior* is a subsequence $w \in \Sigma^+$, optionally constrained to a bounded gap so that the constituent steps occur close together (a serial episode [6]); contiguous behaviors are the zero-gap special case. The *support* of w is the number of traces that contain it,

$$\mathrm{supp}(w) = \left|\{\, m : w \sqsubseteq s(\rho^{(m)}) \,\}\right|,$$

and we write $N(w)$ for its total occurrence count across the corpus. Frequent behaviors, those with $\mathrm{supp}(w) \geq \theta$, are recovered exactly by pattern-growth mining [4, 5]. Crucially, θ is a frequency, not a similarity cutoff: it states how many runs a behavior must appear in to be considered, a quantity with an unambiguous meaning. We do not commit to a single θ; we report the support distribution and let the economic criterion below select. To keep the recovered set tractable and non-redundant we mine *closed* (or maximal) frequent episodes rather than every frequent subsequence, reporting a behavior only when no extension preserves its support; this bounds the output and avoids enumerating every sub-behavior of a frequent one.

We mine the surrounding subsequence, rather than isolated signatures, because the context fixes the step's role: the same generated-query signature occurring within `connect`→`query`→`format` denotes a stable behavior with a well-defined input distribution, and the matched subsequence doubles as the routing condition that tells the agent when the specialist applies. The specialization target is the single model-generated step inside the matched behavior, and $N(w)$ counts that step's in-context occurrences. This distinction, between mining any frequent subsequence and mining one that *contains* a model-generated step, turns out to matter empirically (§6.2).

4.3 The Specialization Trigger: an Economic Criterion

Frequency makes a behavior a *candidate*; economics decides whether to act. Replacing a model-generated step with a specialist does not eliminate its cost, it changes the cost's shape: the agent runs the specialist, screens its output with an online check, and falls back to the frontier when the check rejects it. Writing c_{call} for the frontier cost of the step, c_{spec} for the specialist's per-invocation cost *including any serving overhead*, c_{ver} for the per-invocation cost of the online check, and $\alpha \in [0,1]$ for the rate at which that check accepts the specialist's output (so $1-\alpha$ is the fallback rate), the expected cost of the specialized path is

$$c_{\mathrm{path}} = c_{\mathrm{spec}} + c_{\mathrm{ver}} + (1-\alpha)\,c_{\mathrm{call}},$$

and the expected saving per invocation is

$$\Delta(w) = c_{\mathrm{call}} - c_{\mathrm{path}} = \alpha\,c_{\mathrm{call}} - c_{\mathrm{spec}} - c_{\mathrm{ver}}.$$

The frontier cost is recovered only on the fraction α the specialist gets right, while the run-and-check cost is paid every time; the specialist's accuracy thus enters the economics directly, where an accuracy-free saving $c_{\mathrm{call}} - c_{\mathrm{spec}}$ would overstate it. In deployment this online check is typically a cheaper integrity test than the correctness oracle used at

training time, since a novel input has no reference output: it catches malformed or post-condition-violating outputs rather than certifying full correctness, so α is an acceptance rate under that test and the residual of accepted-but-wrong outputs is a separate risk (§8).

Let c_{train} be the one-time cost of producing a *deployable* specialist (dataset assembly and verification, training runs including failed attempts, the gating evaluation, and the engineering and infrastructure cost of standing the specialist up and maintaining it), and $\hat{N}(w)$ the projected number of future invocations. A model-generated behavior is worth compiling when its total saving clears that one-time cost,

$$\hat{N}(w)\,\Delta(w) > c_{\text{train}} \quad \text{and} \quad g(w) = 1,$$

with break-even count

$$N^{\star} = \frac{c_{\text{train}}}{\alpha\, c_{\text{call}} - c_{\text{spec}} - c_{\text{ver}}},$$

defined only when the denominator is positive. That positivity is a *feasibility* condition independent of volume:

$$\alpha > \alpha_{\min} := \frac{c_{\text{spec}} + c_{\text{ver}}}{c_{\text{call}}}.$$

A specialist whose accuracy falls below $\alpha_{\min}$ costs more to run and check than it saves, and no frequency redeems it; a behavior can be both frequent and model-generated yet still not worth specializing if no specialist clears $\alpha_{\min}$. We stress that c_{train} is dominated in practice by human and infrastructure cost, not by training compute, and that $N^{\star}$ scales linearly in it: an order-of-magnitude error in c_{train} is an order-of-magnitude error in the break-even, so this is the term a deployment must estimate most carefully (§6.3).

Two further refinements keep the rule honest. First, $\hat{N}(w)$ is a forecast about a behavior that may stop recurring, so we do not act on a raw point estimate: we require a margin, $\gamma\hat{N}(w) > N^{\star}$, with a discount $\gamma \in (0,1]$ that down-weights the forecast for persistence risk and the time value of the committed compute (calibrated to the measured drift of §7). Second, caching and specialization are distinct responses keyed on different things. A cache keys on *input repetition*, and we detect it with the same discrete device used everywhere else: applying κ to the step's input as well as its operation, a cache hit is simply the event that the input's signature has been seen before, so the stored output can be replayed with no model at all. Where a step's inputs collapse onto a small set of signatures, a cache strictly dominates a specialist; caching LLM-step outputs is itself standard [13], and keying on signature equality rather than embedding distance keeps the decision consistent with the rest of the method. Specialization is for the complementary case, inputs whose signatures *vary* widely while the underlying skill is shared, where no finite cache suffices but a small model generalizes across the variation. The trigger above governs that second case; the signature-repetition check is applied first, and only steps whose inputs vary are passed to the break-even.

This is the entire decision rule, and it contains no notion of similarity: discrete signature equality determines what counts as a behavior, frequency determines candidacy, an input-repetition check separates caching from specialization, and a cost inequality denominated in compute determines action. The threshold is meaningful for a given workload rather than chosen by hand.

4.4 Verified Harvesting

For a behavior w selected by the criterion, the corpus already contains every instance the agent has executed. We collect the input/output pairs realized at its model-generated step,

$$H(w) = \big\{(x_k, y_k)\big\}_{k=1}^{N(w)},$$

where x_k is the step's input (the natural-language request and context) and y_k the output the agent produced (the generated SQL, the extracted record). We then screen these pairs with an automatic verifier and retain only those that pass,

$$\tilde{H}(w) = \big\{(x_k, y_k) \in H(w) : V(x_k, y_k) = 1\big\}.$$

Where an execution oracle exists, V is that oracle: the query executes and matches a reference result, the code passes its tests [17, 18]. Where no execution oracle exists, V may be an LLM judge [19] applied *only to these concrete pairs*, as a correctness screen on specific outputs. The judge never decides whether two behaviors are the same; that decision was made discretely in §4.1. Keeping this boundary sharp is what prevents the non-reproducibility of similarity judgments from leaking back into the method. The verified set $\tilde{H}(w)$ is the training data for the specialist, and the same V later gates whether the trained specialist is allowed to serve, so a single oracle cleans the data and guards deployment. The deployment studied in §6.5 runs exactly this judge layer in production, which lets us confirm the design is operable even though a clean yield figure is not yet measurable.

A specialist is then a small base model adapted by LoRA [11] and trained on $\tilde{H}(w)$; it is admitted only if it matches the frontier on held-out verified pairs from the same behavior. We treat the training and gating mechanics as downstream of discovery and out of scope for this paper, whose contribution is the discrete, cost-aware discovery and the verified harvest that feeds them.

5 Why Discrete, Not Vector

It is worth stating plainly why the pipeline avoids embeddings, because the temptation to vectorize is strong and, we argue, mistaken for this problem. The empirical collapse reported in §6.1 is the concrete payoff of the argument that follows.

A discovery method's job is to report a property of the workload: how concentrated agent behavior is, and which behaviors dominate. That property is only well-defined if "the same behavior" partitions traces into groups. Signature equality does: it is reflexive, symmetric, and transitive, an equivalence relation by construction, so it carves the trace

corpus into disjoint behaviors and the concentration statistic ("80% of work falls into ten behaviors") is unambiguous. Thresholded embedding similarity is *not* an equivalence relation: *a* can lie within the cutoff of *b* and *b* of *c* while *a* and *c* are far apart, so similarity alone does not define groups at all. Recovering groups from it requires a further clustering or linkage rule, itself carrying arbitrary choices, and different linkages partition the same traces differently. The headline number is then not a fact read from the workload but an artifact of an encoder, a threshold, and a linkage, none of which the problem fixes.

Even setting the partition aside, an embedding offers no *account* of why two operations were merged. A cosine distance is a bare scalar; it cannot tell an auditor that two queries were judged the same because they differ only in a literal. Canonicalization can: the merge follows from inspectable rules stating which differences are irrelevant (the literal in a query, the URL in a clone). For a system that spends money autonomously on the strength of "this behavior recurs," an auditable reason is worth more than a number. Embedding distance is also ungrounded in execution: two queries differing only in a constant are functionally identical yet may embed far apart, while superficially similar calls may do entirely different work. Canonicalization keys on exactly the structure that determines behavior and normalizes away the rest.

The design freedom that remains, the granularity of κ, is not hidden inside an opaque scalar but exposed as the ladder $\kappa_0 \preceq \cdots \preceq \kappa_3$, and we report how repetition changes along it (§6.1). Both a fixed encoder and a fixed κ are reproducible in the trivial sense that re-running them repeats the output; the difference is that κ is a short list of readable rules an independent party can not only replicate but *audit and contest*, whereas an encoder's decision boundary is uninspectable. Inspectable reproducibility, not reproducibility as such, is the property discreteness buys, and it is the reason discreteness is a requirement here rather than a stylistic preference.

6 Empirical Validation on Production Traces

We instantiate the method on real agent traces and report what they show, at the strength the data supports. The corpus is 57,939 captured agent-execution trace events (total recorded frontier spend $5,198.90) from a production observability platform; we focus the analysis on a single high-volume coding-agent workload of 1,785 traces ($3,941.80 of frontier spend, a median of 16 operations per trace, 49 distinct tools, 17,854 model-generated steps, and 19,562 tool operations). Every figure is read directly from the trace table under a fixed, inspectable κ.

6.1 Concentration Along the Ladder

We compute the canonicalization ladder over the 19,562 tool operations of the workload, each level strictly coarsening the previous (Table 1). The SQL is given in the appendix so the merges can be audited rather than trusted.

Table 1: Distinct signatures $|\Sigma|$ over 19,562 tool operations as the canonicalization κ coarsens. Each level strictly nests the previous. The space collapses by 46× before any normalization and to 49 operation skeletons, with no encoder and no distance threshold anywhere.

Level	Definition	$\lvert\Sigma\rvert$
κ_0	exact `tool_name` + raw input	422
κ_1	literals normalized (#, `S`, ws)	119
κ_2	+ paths/punctuation stripped	59
κ_3	`tool_name` only (skeleton)	49

The space collapses 46× at κ_0, *before any normalization*, and monotonically to 49 skeletons along the ladder, exactly the behavior §5 predicts. At the skeleton level the top 9 of 49 operation types cover 94.6% of all calls. This is the paper's concentration premise read from traces rather than asserted, and it is the concrete form of the anti-embedding argument: equivalent operations become *literally equal* under κ and the space collapses by orders of magnitude with no encoder to tune and no threshold to defend. The monotonicity of Table 1 is the empirical content of the ladder claim of §4.1; concentration is already extreme at κ_0, so the result does not depend on aggressive normalization, which answers the obvious worry that the collapse is an artifact of coarse canonicalization.

We are precise about the scope of this measurement, because it is easy to over-read. This is concentration of *single typed operations* (length-one behaviors in the sense of §4.2). It is the premise the method rests on and it is necessary background for specialization, but it is not by itself a measurement of how often a multi-step, *model-generated* behavior recurs, and most of the operations driving the 94.6% are deterministic file and shell tools, not model-generated steps. The recurrence that the economic trigger actually consumes is measured separately, and incompletely, below.

6.2 Frequent Subsequences

Mining consecutive `tool_name` bigrams within a trace (support = distinct traces containing the bigram, out of 1,785) recovers the canonical edit→write→run→inspect loop of a coding agent (Table 2). The same short subsequences recur across ∼12% of all traces over a 49-symbol alphabet despite each underlying task differing: the "different surface request, identical behavior" structure the method is built to find. This validates that pattern-growth mining recovers genuine recurring structure from the trace corpus.

Two honest qualifications keep this from being read as more than it is, and the second is the most important gap in the present study. First, the subsequences recovered here are transitions between *deterministic* tool steps; under the paper's own framing (§4.3) these are caching candidates, not specialization candidates, since they carry no model-generated step. They are the right validation of the *mining machinery* and the wrong objects for the *economic trigger*.

Table 2: Top consecutive-bigram behaviors by trace support (of 1,785 traces). Mining recovers the canonical coding-agent loop. *These subsequences are transitions between deterministic tool steps and carry no model-generated step*; under §4.3 they are caching candidates, not specialization candidates. They establish that the mining machinery works, not that a specializable behavior recurs.

Behavior (bigram)	occ.	support
edit → write_file	246	12.9%
execute → run_command	241	12.8%
run_command → edit	237	12.7%
read_file → think	232	12.3%
execute → ls	231	12.2%
ripgrep → execute	228	12.1%

Table 3: Per-call cost of the model-generated step by model, same workload. A frontier tier at \$0.30–\$0.82/call sits alongside a cheap tier at \$0.012–\$0.019/call. The cheap tier is the price a specialist must hit; the spread is the headroom the trigger trades against c_{train}.

Model (model-gen. step)	calls	total \$	\$/call
opus-4.7	2,675	1932.21	0.722
gpt-4o	1,084	883.62	0.815
opus-4.6	2,808	840.90	0.300
sonnet-4.6	2,417	139.78	0.058
glm-5.1	3,412	59.89	0.018
kimi-k2.6	2,767	52.34	0.019
gpt-5.4-mini	1,192	14.43	0.012

Second, and consequently, the specialization-relevant quantity is the in-context recurrence $N(w)$ of a behavior that *contains* a model-generated step, which requires mining episodes over the full event stream, including the model-generated rows the present pull excludes, rather than the tool-only bigrams shown here. We therefore do *not* report a measured $N(w)$ for any real specializable behavior, and we treat closing this gap as the single highest-value next measurement (§7).

6.3 Real Economics of the Model-Generated Step

The model-generated step ($n = 17{,}854$) costs a real median of \$0.0247 and mean of \$0.2208 per call (p90 \$0.83, max \$11.57), against a cheap-model tier present *in the same workload* at \$0.012–\$0.019 per call, a 40–60× spread between frontier and cheap tiers (Table 3). This is exactly the gap a specialist priced at the cheap tier is meant to capture, and it confirms the cost regime the economics of §4.3 assume is real rather than stipulated. The illustrative \$0.011 of §6.4 is, if anything, conservative on the frontier side.

Recomputing the break-even at the real *median* $c_{\text{call}} =$ \$0.0247 ($\alpha = 0.95$, $c_{\text{spec}} =$ \$0.0010, $c_{\text{ver}} =$ \$0.0003) gives $\Delta \approx$ \$0.0222 and, at a compute-only $c_{\text{train}} =$ \$80, $N^\star \approx$ 3,600 invocations, with feasibility floor $\alpha_{\min} \approx 0.053$ cleared comfortably. Two cautions keep this from being read as a favorable result, and both follow directly from §4.3.

(i) The \$80 prices only compute. As §4.3 states, c_{train} is dominated by the engineering and infrastructure cost of standing up and maintaining a per-behavior specialist, not by training runs. An honest c_{train} is plausibly one to two orders of magnitude larger, moving $N^\star$ into the 36,000–360,000 range. The real higher c_{call} *lowers* $N^\star$ and the realistic higher c_{train} *raises* it; these corrections partly offset, and the defensible statement is that the cheap-tier-replacement case remains plausibly positive on this workload, not that it is markedly favorable.

(ii) The mean is the wrong statistic here. The mean $c_{\text{call}} =$ \$0.22 (which would give $N^\star \approx 370$) should not be used for this purpose. The expensive calls carry very large context (mean ∼38k input tokens) and are the agent's main reasoning steps (precisely the calls a small specialist *cannot* replace), so the workload mean attributes the cost of un-specializable calls to specializable ones. Because the present pull does not isolate the cost of a *specific* recurring model-generated behavior, every $N^\star$ here is a workload aggregate, not a per-behavior break-even; the per-behavior figure awaits the episode mining of §6.2/§7.

6.4 Illustrative Discrimination of the Rule

The measured costs above fix the regime; they do not, by themselves, show the decision rule *discriminating* across the full set of outcomes it must produce (specialize, cache in two kinds, defer, and frontier), because the present corpus does not isolate five real behaviors with measured recurrence, accuracy, and verifiability. To show the rule discriminates we apply it to five *constructed* candidate behaviors at the measured median cost (Table 4); the only hypothetical inputs are each behavior's $\hat{N}$, α, and verifiability, which the pull does not yet provide. **The $\hat{N}$, accuracy, and verifiability columns are illustrative; the per-call costs and $N^\star$ are anchored to the measured median.**

At $c_{\text{call}} =$ \$0.025 the per-invocation saving is

$$\Delta = 0.95(0.025) - 0.0010 - 0.0003 \approx \$0.0225,$$

and the break-even is $N^\star = 80/0.0225 \approx 3{,}600$ invocations (compute-only c_{train}; see §6.3(i) for the realistic figure). The criterion separates the five cases cleanly: a frequent, model-generated, verifiable behavior with *varying* inputs is compiled; a model-generated, verifiable behavior whose inputs collapse onto a few signatures is memoized rather than specialized, because a cache dominates when there is nothing to generalize; a frequent but deterministic step is cached as an I/O optimization, not replaced by a model; a model-generated behavior with no available verifier is deferred, since it cannot be safely harvested; and a genuinely rare behavior is left on the frontier because it never repays training.

Three points generalize beyond the specific inputs. First, frequency alone is not the criterion: the deterministic `fetch→parse` step is the most frequent yet is never a model cost, so a cache, not a specialist, is the right tool, and this is not merely hypothetical, since the bigrams of Table 2 are exactly such deterministic high-frequency steps. Second, caching pre-empts specialization wherever input signatures

Table 4: The specialization rule applied to five constructed behaviors at the measured median cost ($N^\star \approx 3{,}600$). "Gen." is g; "Ver." marks whether a verifier exists; "Inp." marks whether the step's inputs, under κ, *vary* or *rep.*eat. The two *cache* rows differ in kind: `query[fixed]` memoizes a model-generated step whose inputs repeat; `fetch`→`parse` caches a deterministic call. $\hat{N}$, accuracy, and verifiability are illustrative; costs and $N^\star$ are measured.

Behavior	$\hat{N}$/yr	Gen.	Ver.	Inp.	Decision
connect → query[agg] → format	18,000	yes	yes	vary	**specialize**
query[fixed] → format	12,000	yes	yes	rep.	cache
fetch[api] → parse	95,000	no	n/a	n/a	cache
search → summarize	11,000	yes	no	vary	defer
query[rare skel.]	500	yes	yes	vary	frontier

repeat: `query[fixed]` clears both gates but its inputs reduce to a few signatures, so memoizing dominates training. Third, verifiability is a hard gate: `search`→`summarize` clears break-even but cannot be harvested cleanly, so under our scoping it is deferred. Even the compiled behavior saves only $\hat{N}\Delta \approx \$400$/yr before its one-time cost; per-behavior savings are modest by construction, and the case for the method rests on the *tail* of many such behaviors across a workload and their reuse across tenants (neither of which a single workload demonstrates), together with latency gains this monetary accounting does not credit.

6.5 Verification Layer

The platform natively operates the paper's verifier/judge design: 18 named behaviors, 21 LLM judges bound to those behaviors and run on *cheap* models (haiku-class and mini-class), and 894 recorded binary behavior-verdicts, each carrying a pass/fail, a reason, and the judge and behavior identity. This is an LLM judge confined to screening concrete outputs, never deciding behavioral identity, exactly as §4.4 requires, and its existence in production confirms the harvest design is operable. A clean harvest-yield number ($|\tilde{H}|/|H|$) requires oracle-backed judges run over complete traces; the present verdict rows do not support that measurement, so we report the mechanism as implemented and leave the quantitative yield to §7 rather than report a figure that cannot yet be cleanly measured.

6.6 Stability

Across the three-month window the tool alphabet stays within a 16–40 signature range, consistent with a stable signature space and infrequent re-mining. Two limits on this cut are worth stating: the platform changed from per-trace to long-session grouping partway through the window, so per-trace counts are *not* comparable across months; and alphabet stability is a weaker property than stability of *behaviors and their supports*, which is what the discount γ of §4.3 actually depends on and which remains to be measured over comparable windows.

6.7 What This Validation Does and Does Not Establish

It establishes the method's premise and the front of its machinery on real traces: discrete canonicalization collapses the operation space by orders of magnitude ($46\times$ at κ_0, to 49 skeletons), frequent-subsequence mining recovers genuine recurring structure, the cost regime is real and shows the frontier-to-cheap spread the economics assume, and the verified-harvest design corresponds to an operational judge layer. It does *not* yet establish the payoff: it does not measure the in-context recurrence $N(w)$ of a model-generated behavior (the bigrams mined here are deterministic), so it does not compute a break-even hit rate over real specializable behaviors; it does not measure harvest yield; and it does not train a specialist and show it matching the frontier at reduced cost. These are the remaining cuts of §7, and we present them as open rather than answered.

7 Remaining Measurements

The validation above fills the premise; four measurements would close the loop from discovery to demonstrated value, and we specify each so it can be run (or used to falsify the method) directly against the trace schema of §6.

Per-behavior recurrence and break-even hit rate. The highest-value next step, and the one that turns §6.3's aggregate $N^\star$ into a real result: mine *episodes that contain a model-generated step* over the full event stream (not the tool-only bigrams of §6.2), count each such step's in-context occurrences $N(w)$, and report the fraction of high-support, model-generated behaviors whose projected $\gamma\hat{N}(w)$ clears $N^\star$ at the measured per-step cost and a realistic c_{train}. This is the measurement that says how often discovery actually fires, and it is presently the chief gap.

Harvest yield. For triggered behaviors, run oracle-backed verifiers over complete traces and report $|\tilde{H}|/|H|$, the verifier-pass rate of the harvested pairs. This quantifies how much clean training data discovery produces and surfaces behaviors where the agent is frequently wrong (low yield). The judge layer of §6.5 is the substrate; what is missing is execution-grounded oracles over full traces.

Downstream effect. For a sample of triggered behaviors, train the LoRA specialist on $\tilde{H}$ and report its verified accuracy against the frontier on held-out pairs and the realized cost reduction net of discovery, harvesting, and training. This is the end-to-end claim the present paper deliberately does not make.

Stability of behaviors. Report how discovered behaviors and their supports, not merely the tool alphabet (§6.6), drift across comparable time windows, which both indicates re-mining cadence and calibrates γ.

Reporting these, each read from traces under a fixed, inspectable κ, is what would carry the method from a validated premise to a demonstrated end-to-end result.

8 Limitations

The empirical study is on a *single* coding-agent workload, so the concentration magnitudes of §6.1, while striking, are an existence proof on a real agent rather than a population-level result, and a single workload cannot exhibit the cross-tenant reuse on which the method's aggregate case partly rests. The canonicalization κ is a design choice, and different choices yield different behaviors and supports; we mitigate this by exposing the choice as a reported ladder (Table 1) rather than burying it, but a poorly designed κ can over- or under-merge. The most consequential gap is that the present pull mines only deterministic tool subsequences (§6.2), so we report no measured recurrence $N(w)$ for a specializable model-generated behavior and therefore no real break-even hit rate; the $N^\star$ figures of §6.3 are workload aggregates, they assume a compute-only c_{train} whose realistic value is one to two orders of magnitude larger, and the favorable mean-based figure is rejected in-text as a selection artifact. Verified harvesting requires an automatic correctness oracle, which scopes the method to verifiable behaviors (SQL, code, schema-checkable extraction, tool calls with checkable post-conditions); for open-ended behaviors with no oracle, LLM-judge screening is a weaker substitute, and we prefer to defer such behaviors than to train on unverified data; the operational judge layer of §6.5 screens outputs but does not yet supply a clean yield figure. The cost accounting also does not price the downstream cost of specialist outputs that pass the cheaper online check but are subtly wrong, since at inference time on a novel input there is no reference to catch them, and α measures passing that check rather than ground-truth correctness; a poor-precision online check would ship wrong answers to save sub-cent amounts, a value-proposition risk and not merely an accounting omission. The break-even further acts on a forecast $\hat{N}$ discounted by γ, both calibrated to drift that §7 still has to measure on comparable windows. Finally, no specialist is trained in this paper: the end-to-end demonstration (a compiled behavior matching the frontier at reduced realized cost) is future work, and the illustrative columns of §6.4 must not be read as findings; the measured quantities of §6 are the findings, and they are the premise, not the payoff.

9 Conclusion

We have described how an agent can discover, on its own, which of its behaviors recur often enough to be worth compiling into cheap specialists, and how to do so without the arbitrariness that embedding-based similarity would introduce. The method is discrete throughout: typed operations are reduced to canonical signatures so that equivalent behaviors become equal rather than merely close; behaviors are frequent subsequences recovered by exact pattern mining; the decision to specialize is an economic break-even on a behavior's model-generated step, net of the specialist's accuracy and the cost of verifying and falling back; and training data is harvested along discovered behaviors and screened by a verifier, with an LLM judge confined to screening concrete outputs and never to judging behavioral identity. On real production traces the premise holds with room to spare: typed operations collapse 46× under discrete canonicalization with no encoder, mining recovers the canonical work loop, and the model-generated step shows the frontier-to-cheap cost spread the economics require, while the payoff (a per-behavior break-even, a harvest-yield figure, and a trained specialist matching the frontier) is left as the explicit, specified next step rather than asserted. The discovery layer is independent of how the resulting specialists are trained or served, and it is deliberately scoped to behaviors that admit an automatic correctness oracle, which already covers a broad and growing class of agent work in code, data, and structured tool use. The same discrete signal that lets an external pipeline harvest and compile these behaviors is, in principle, what an agent would use to detect a recurring behavior online and trigger training of its own small specialist; building that closed self-improving loop, and carrying the present validation from premise to end-to-end result, is the natural next step.

References

[1] Bala, V., Duesterwald, E., & Banerjia, S. (2000). Dynamo: A transparent dynamic optimization system. *PLDI*.

[2] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

[3] Chu, S., Wang, C., Weitz, K., & Cheung, A. (2017). Cosette: An automated prover for SQL. *CIDR*.

[4] Agrawal, R., & Srikant, R. (1995). Mining sequential patterns. *ICDE*.

[5] Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., & Hsu, M.-C. (2004). Mining sequential patterns by pattern-growth: The PrefixSpan approach. *IEEE TKDE*, 16(11), 1424–1440.

[6] Mannila, H., Toivonen, H., & Verkamo, A. I. (1997). Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3), 259–289.

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

[8] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *NeurIPS*.

[9] Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., et al. (2023). DSPy: Compiling declarative language model calls into self-improving pipelines. arXiv:2310.03714.

[10] Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the knowledge in a neural network. arXiv:1503.02531.

[11] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA: Low-rank adaptation of large language models. arXiv:2106.09685.

[12] Chen, L., Zaharia, M., & Zou, J. (2023). FrugalGPT: How to use large language models while reducing cost and improving performance. arXiv:2305.05176.

[13] Bang, F. (2023). GPTCache: An open-source semantic cache for LLM applications. *Proc. NLP-OSS Workshop at EMNLP*.

[14] Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., & Hajishirzi, H. (2023). Self-Instruct: Aligning language models with self-generated instructions. *ACL*.

[15] Zelikman, E., Wu, Y., Mu, J., & Goodman, N. D. (2022). STaR: Bootstrapping reasoning with reasoning. *NeurIPS*.

[16] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., et al. (2023). Self-Refine: Iterative refinement with self-feedback. *NeurIPS*.

[17] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., et al. (2021). Training verifiers to solve math word problems. arXiv:2110.14168.

[18] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., et al. (2021). Evaluating large language models trained on code. arXiv:2107.03374.

[19] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., et al. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *NeurIPS*.

[20] Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., et al. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL. *EMNLP*.

[21] Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., et al. (2023). Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs (BIRD). *NeurIPS*.