WHEN SHOULD AGENTS BUILD REUSABLE TOOLS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern agents can improve over time by turning solved tasks into persistent reusable tools, but building such capabilities requires an upfront investment that may not pay off. We study when an online agent should build a new tool rather than patch the current task or reuse an existing one, under uncertain task recurrence and stochastic tool quality. We formulate this as cost-sensitive capability acquisition and develop Demand-Sensitive Acquisition (DSA), a controller that combines recurrence estimates with uncertainty in current success, durable admission, and future reuse. In a separable model where each tool serves one recurring task type, we derive the exact finite-horizon optimal policy and decompose the cost gap of building when a task type first appears; on an executable API-migration benchmark with 108 LLM-generated candidates, DSA lowers modeled cost by 6.7% over 200 heavy-tailed task sequences, relative to a pattern-level baseline that builds when each pattern first appears. These results frame agent self-improvement as a sequential investment problem over reusable capabilities.

1 INTRODUCTION

Modern AI agents increasingly convert experience into persistent resources—tools, executable skills, programs, and structured memories—that can be reused across future tasks (Cai et al., 2024; Wang et al., 2023; Dai et al., 2026). Such persistence can amortize expensive reasoning or generation over many later interactions, but creating and validating a reusable capability is itself costly. An agent therefore faces a decision that is distinct from tool generation alone: even when it *can* build a useful tool, should it pay to build that tool *now*, rather than solve the current task locally or reuse what it already has? Building too readily wastes acquisition effort on rare tasks, while delaying a worthwhile acquisition can repeatedly incur local service costs and forfeit future reuse. This paper studies this decision problem: *when should an agent invest in a persistent reusable capability?*

API migration provides a concrete instance of this tradeoff. A breaking library update can affect a stream of downstream client programs. For each broken client, an agent may patch the program once, reuse an already available migration tool, or spend additional effort to generate and validate a reusable transformation. If only a few clients will ever require the migration, the reusable tool may never repay its upfront cost; if the same migration recurs frequently, repeated patching can be much more expensive than acquiring a durable transformation early. Tool creation is also uncertain: a generated transformation may fail, or it may successfully repair the triggering client yet fail broader validation required for future reuse. Even a rejected first candidate does not imply that a later attempt would be uneconomic. Recent work shows that LLMs can synthesize reusable program transformations and evaluate their transfer across examples or projects (Ramos et al., 2026; Reyes et al., 2026a); our focus is the complementary pre-build question of whether uncertain future reuse justifies paying for generation and validation in the first place.

This question is related to, but not resolved by, existing mechanisms for persistent agent improvement. Tool-making systems already amortize creation effort across repeated use (Cai et al., 2024); memory and skill systems use recurrence, performance, or learned library-management signals to decide when to consolidate or update persistent resources (Dai et al., 2026; Li et al., 2026; Wei et al., 2026). Here we isolate a different decision-theoretic object. Before acquisition, a candidate's outcome is unknown; patching and existing tools remain available alternatives; success on the current task need not imply validation for future reuse; and a failed validation need not make a later attempt unattractive. These distinctions turn tool creation into a sequential acquisition problem rather than a one-time generation decision.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The acquisition decision is difficult because several uncertainties interact. Future value depends on how often related tasks recur over the remaining horizon, whether generation produces a reusable tool, and what service is already available from the current library. In addition, the information sources are asymmetric. Task arrivals reveal demand without requiring an acquisition, whereas evidence about a new tool's reusable quality is often obtained only after generation and validation. Therefore, neither recurrence nor candidate quality alone determines whether acquisition is worthwhile. At a high level, acquisition is worthwhile when the expected current-task saving plus the admission-weighted value of remaining reuse exceeds the one-time build cost.

Summary of Contributions. We formulate persistent tool creation as a finite-horizon patch–reuse–build problem, explicitly separating current-task success from validation for future reuse. This allows a generated candidate to help immediately without becoming a persistent capability and allows later attempts after unsuccessful validation. Building upon this, our main contributions are threefold.

(1) Demand-Sensitive Acquisition (DSA). We develop DSA, an online controller that combines recurrence estimates with candidate-quality beliefs, durable admission, and future reuse to decide whether an eligible acquisition justifies its cost. DSA updates its beliefs from observed task arrivals, build outcomes, and executed reuse, while using a context-dependent prior to represent systematic variation in candidate quality.

(2) Exact acquisition decision theory. In a known-parameter separable model, where each reusable capability serves one recurring task type, we derive a closed-form acquisition score whose sign characterizes the exact finite-horizon optimal action. We also derive the exact cost gap of a heuristic that builds once when a task type first appears and never retries after rejection, separating premature acquisition from the cost of abandoning a capability whose future value remains positive.

(3) Executable evaluation. We evaluate DSA in a controlled API-migration environment with 108 frozen, one-shot LLM-generated LibCST transformation candidates across 12 API families and multiple syntax patterns. Under heavy-tailed recurrence, DSA reduces modeled cost by 6.69% relative to a pattern-level baseline that builds when each pattern first appears; controlled demand and cost studies show that the value of selective acquisition is regime dependent.

Taken together, our results suggest that persistent self-improvement should be viewed not only as a generative problem, but also as an investment-control problem. Improvements in tool generation do not eliminate the need to decide which opportunities justify acquisition effort or when an unsuccessful acquisition should be tried again. The exact theory isolates the timing and retry structure, while the executable study tests whether the same decision principle is useful with learned demand, stochastic candidate quality, operational scope restrictions, and a finite tool pool.

The rest of the paper formalizes the acquisition problem in Section 3, presents DSA in Section 4, develops the exact decision theory in Section 5, and evaluates its behavior and cost in Section 6. The appendix provides the formal model and proofs, complete experimental specifications and extended results, and broader discussion.

2 Related Work

We place our work at the intersection of persistent agent capabilities, reusable program transformations, and online acquisition and investment.

Persistent agent capabilities and library management. Agents increasingly maintain tools, skills, or memories for future reuse. Cai et al. (2024) explicitly motivate tool making through amortization, while Wang et al. (2023) maintain a persistent executable skill library. More recent systems make library updates themselves adaptive: recurrence can trigger memory consolidation (Dai et al., 2026), learned policies can manage skill extraction and retention (Li et al., 2026), and interacting skills and tools can be jointly evolved and retired (Wei et al., 2026). These works establish that agents already make creation and maintenance decisions. We instead formalize the pre-acquisition investment question: whether uncertain current and future service value justifies paying to generate and validate a new executable capability.

Reusable program transformations and API migration. LLMs can generate reusable migration artifacts rather than independent one-off fixes. Ramos et al. (2026) synthesize testable migration scripts, and Reyes et al. (2026a) generate AST transformation rules and evaluate replay and transfer.

Related work also mines or synthesizes transformation procedures from software changes (Ramos et al., 2023; Cummins et al., 2024). This literature motivates our executable setting and already demonstrates that local correctness need not imply reusable validity. Our complementary question arises *before* reusable quality is known: when is it worth incurring acquisition cost at all? Further transformation and repair work is reviewed in the appendix.

Online acquisition and investment. Our decision problem is related to rent-or-buy and persistent-resource acquisition. Multislope ski rental trades an upfront payment for lower future operating cost (Lotker et al., 2008), while online facility location makes acquisition value depend on existing service coverage (Meyerson, 2001). Real-options models emphasize the value of waiting under investment uncertainty (McDonald & Siegel, 1986; Pindyck, 1993). In online learning, Kalvit & Zeevi (2022) study endogenous arm acquisition. The closest online-learning precursor involving persistent generated actions, Xu et al. (2026b), studies costly generation and reuse and develops an optimistic online-learning rule with regret guarantees. In that model a generated action is tailored to the triggering context; here a reliable patch remains available, success on the triggering task and validation for future reuse are distinct stochastic outcomes, failed acquisitions can be retried, and recurring capability types admit an exact finite-horizon characterization. We additionally evaluate the acquisition decision with executable program transformations.

3 Problem Setup

Here we formalize the online patch–reuse–build decision and introduce the notation needed in the main paper.

3.1 Problem Formulation

We consider an agent serving T sequential tasks. At time t, it observes the current task, which has a generic task type $X_t \in [K] := \{1, \dots, K\}$, and observable features ξ_t. A build, when considered, occurs in an observable generation context $z_t = \zeta(X_t, \xi_t)$. In the executable benchmark, the observable task description includes an exact API family f_t and coding pattern r_t. For a current family f, each pair (f, k) with $k \in \{\text{direct}, \text{symbol-alias}, \text{module-alias}\}$ is a *service cell*. The agent maintains a persistent library $\mathcal{L}_t$ of admitted tools, each with a declared operational scope specifying the patterns on which the controller may invoke it. Future tasks and the quality of ungenerated tools are unknown.

For each task, the agent chooses among three actions:

1. **Patch** solves the current task at a reliable one-off cost and leaves the library unchanged.
2. **Reuse** applies an admitted tool within its declared scope; detected failure falls back to patch.
3. **Build** pays to generate and validate one candidate tool for the current task.

A build may solve the current task and may, separately, pass validation and enter the persistent library. Admitted tools can serve later tasks; rejected candidates are not retained, and a later eligible task may trigger another attempt.

The agent minimizes cumulative serving and acquisition cost. Patching is local and reliable, whereas building pays a larger upfront cost for a capability whose value depends on recurrence and generation quality. In our analysis, tasks are recurring abstract types; in the executable benchmark, they are client programs affected by API changes and reusable tools are LibCST transformations.

3.2 Key Quantities

Two pieces of notation recur throughout the paper. First, $\mathcal{L}_t$ denotes the tools available before the decision at time t. Second, each build has distinct *current* and *persistent* consequences.

Definition 1 (Current success and durable admission)**.** *For a build in context z, let $G_z \in \{0, 1\}$ indicate whether the candidate serves the current task, and let $I_z \in \{0, 1\}$ indicate whether it is admitted to the persistent library. We require $I_z \le G_z$, allowing failure, current-task success without admission, and current-task success with durable admission.*

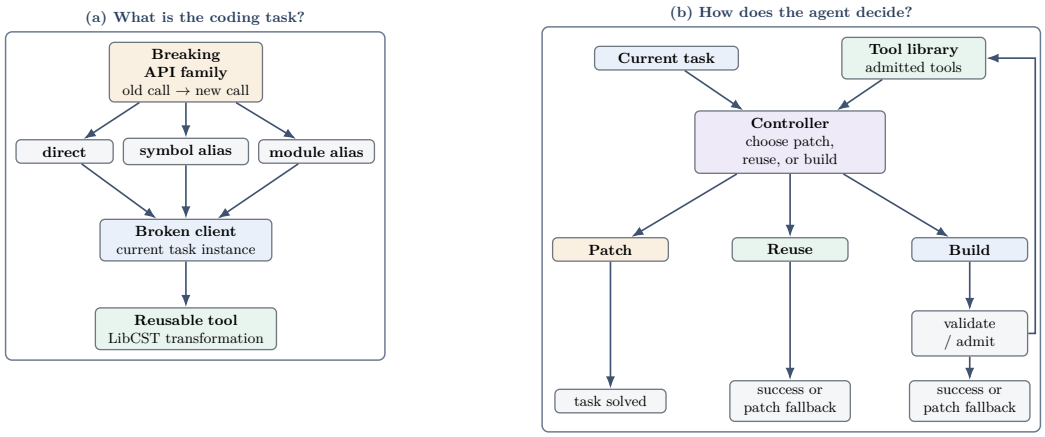


Figure 1: **Problem setup.** (a) A breaking API family can recur across client programs through different observable coding patterns. Each arriving task is a broken client instance, while a reusable tool is an executable LibCST transformation. (b) For each task, the controller chooses a one-off patch, reuse of an admitted tool, or acquisition of a new candidate. A candidate can serve the current task without being admitted for future reuse; admitted tools enter the persistent library, and unsuccessful service attempts fall back to patching.

Let σ_z be the saving relative to patching when current service succeeds, so the realized current-task saving is $R_z := G_z\sigma_z$. Let b_z be the acquisition and validation cost and $q_z := \Pr(I_z = 1)$ the admission probability. Future service quality, demand, and posterior quantities are introduced where needed.

Definition 2 (Cumulative modeled cost)**.** *For a causal policy π, let $J_T(\pi)$ be its expected cumulative cost over T tasks, including patching, tool application and fallback, and paid build and validation attempts. The objective is to minimize $J_T(\pi)$ using only information available at the current decision.*

Scope. We study persistent, task-triggered acquisition: tools are built in response to arriving tasks, admitted tools remain available, and patching can always complete the task. The controller uses only information revealed by arrivals and its own service, generation, and validation outcomes. We do not model proactive off-task generation, maintenance or retirement, or tool composition that changes the generator itself. The exact analysis in Section 5 abstracts these operational details into separable capability types. Section 5.4 states the correspondence and the benchmark departures used in our interpretation. Neither an API family nor a family–pattern pair in the executable benchmark is a direct instance of that abstraction: admitted transformations can transfer across patterns, declared scope can differ from latent executable coverage, and the candidate pool is finite. The theory therefore isolates acquisition timing and retry, while the benchmark tests the same decision principle in a richer operational setting.

4 Method

This section instantiates the patch–reuse–build problem as Demand-Sensitive Acquisition (DSA), the scope-gated posterior-mean acquisition controller used in our executable API-migration study.

4.1 Controller Overview

At each task arrival, DSA observes the API family f_t and coding pattern r_t, the admitted library, and outcomes revealed by previous actions. It chooses among patching, reusing an admitted tool whose declared scope contains the current pattern, or acquiring one new candidate. Its state contains only observable evidence: past task arrivals, outcomes of attempted acquisitions, and reliability evidence

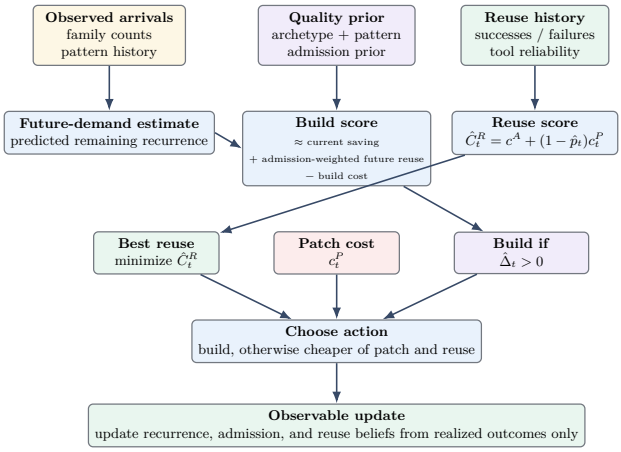


Figure 2: **Demand-Sensitive Acquisition (DSA).** Observed arrivals estimate future recurrence, context-dependent priors and build outcomes inform candidate quality, and reuse outcomes update admitted-tool reliability. These signals determine whether acquisition is worthwhile relative to patch/reuse service.

from tools actually reused. Future tasks and counterfactual candidate outcomes are never policy inputs; offline candidate-quality information enters only through the cross-fitted quality prior, which is estimated from other API families and excludes the current family.

Figure 2 summarizes the information flow. Past arrivals inform remaining demand; a cross-fitted hierarchical prior together with observed build outcomes informs candidate quality; and executed reuse updates admitted-tool reliability.

4.2 Posterior Value Estimation

Patch and reuse. Let c_t^P be the reliable patch cost. For each applicable admitted tool a, DSA estimates reuse success $\hat{p}_t(a)$ from executed outcomes and assigns service cost $\hat{C}_t^R(a) = c^A(a) + (1 - \hat{p}_t(a))c_t^P$, where $c^A(a)$ is the application cost and detected failure falls back to patch. The no-build action is the cheaper of patch and the applicable reuse options.

Build value. A hard observable scope gate prevents new acquisition when an active same-family tool already claims the current pattern. This is a benchmark operational restriction: once the current pattern is declared covered, DSA does not acquire a replacement there. When the gate is open, no admitted same-family tool is selectable on the current pattern, so the candidate's current-task saving is measured relative to patching.

Let A denote the prospective candidate. In posterior draw m, let $N_k^{(m)}$ be remaining demand for service cell (f_t, k), $(G^{(m)}, I^{(m)})$ the sampled current-success/admission outcome, and $S_{A,k}^{(m)}$ the sampled transfer saving associated with pattern k. DSA computes

$$\Delta_t^{(m)} = G^{(m)}\sigma_{z_t} + I^{(m)} \sum_k N_k^{(m)} S_{A,k}^{(m)} - b_{z_t}, \qquad \widehat{\Delta}_t = \frac{1}{M}\sum_{m=1}^{M} \Delta_t^{(m)}, \tag{1}$$

with $M = 512$ and k ranging over the three coding patterns of the current API family. The score mirrors the "current benefit plus admission-weighted future benefit minus acquisition-cost" structure of the separable rule in Section 5, while DSA estimates the relevant quantities online.

The demand draw samples remaining family and pattern counts from the two-level Dirichlet posterior. A cross-fitted hierarchical prior initializes the joint probabilities of build failure, current success without admission, and durable admission; observed builds update this joint state. The transfer term is constructed from a Beta reliability draw $U_k^{(m)}$ as $S_{A,k}^{(m)} = \left[c_t^P U_k^{(m)} - c^A(A)\right]_+$, with executed reuse updating the corresponding reliability beliefs.

Algorithm 1 Demand-Sensitive Acquisition (DSA)

Require: Frozen hierarchical prior, modeled costs, Monte Carlo size $M = 512$

Initialize empty library $\mathcal{L}_1$ and online demand/quality beliefs
for $t = 1, \ldots, T$ **do**
  Observe family f_t and pattern r_t; update the demand posterior
  Let a_t^{srv} be the cheapest belief-based patch/reuse action
  if an active same-family tool claims r_t **then**
    Execute a_t^{srv} and update reuse beliefs if applicable
    continue
  end if
  Draw M posterior worlds for remaining demand, (G, I), and candidate transfer
  $\widehat{\Delta}_t \leftarrow \frac{1}{M} \sum_{m=1}^{M} \left[G^{(m)} \sigma_{z_t} + I^{(m)} \sum_k N_k^{(m)} S_{A,k}^{(m)} - b_{z_t} \right]$
  if $\widehat{\Delta}_t > 0$ and a candidate remains **then**
    Acquire one candidate; execute current service and fixed validation
    Observe (G_t, I_t) and update the build-quality posterior
    if $I_t = 1$ **then**
      Add the candidate and declared scope to $\mathcal{L}_{t+1}$
    end if
    if $G_t = 0$ **then**
      Complete the task by patch fallback
    end if
  else
    Patch the current task
  end if
end for

The future-value term uses standalone latent transfer across the three pattern cells as a predictive proxy for candidate usefulness. Actual reuse remains restricted by the candidate's frozen declared scope, so latent executable coverage does not itself grant permission to serve another pattern. The score also does not subtract a separate future-overlap term for the existing library. The exact posterior sampling, scope, feedback, and cost rules, together with diagnostics of these approximations, are given in Appendix B.1 and B.9.

Primary Decision Rule. As described in Algorithm 1, DSA builds one candidate when the scope gate is open, a candidate remains in the frozen queue, and the posterior-mean score is strictly positive. Otherwise, with the gate open it patches the current task; an exhausted queue incurs no build cost or quality update. When the gate is closed, DSA takes the cheaper patch/reuse action. Rejection does not permanently disable the family–pattern opportunity, so a later eligible task can trigger another attempt.

5 Theory

This section isolates the acquisition-timing mechanism in a clean separable model and characterizes the exact optimal decision in that specialization.

5.1 Separable Stochastic-Admission Model

For the analysis, we specialize the general setting in Section 3 to a known-parameter model in which each recurring task type has its own reusable capability. This removes cross-type substitution and posterior learning from the dynamic program while retaining the two features central to our application: a build can help the current task without producing a durable tool, and a failed admission can be retried later.

Assumption 1 (Separable stochastic admission)**.** *Task types satisfy* $X_t \overset{\text{i.i.d.}}{\sim} \mathrm{Categorical}(p)$ *with* $p_j > 0$ *and* $\sum_j p_j = 1$*. For each type* j*, a build costs* $b_j > 0$*, has mean current-task saving* $\rho_j := \mathbb{E}[R_j]$*, and is durably admitted with probability* $q_j \in (0, 1]$*. Once admitted, the tool permanently saves* $s_j > 0$ *on future type-*j *tasks and has no value on other types. Build actions are task-*

triggered: at most one build may be attempted in a period, and only for the currently arriving type. Build attempts are fresh and may be retried after non-admission; all parameters are known for this decision-theoretic analysis, and $\rho_j - b_j \le s_j$ so rebuilding an already active identical capability is never profitable.

The current-success variable G_j enters this known-model analysis through $\rho_j = \mathbb{E}[R_j]$; durable state transition is governed separately by q_j. The known-parameter assumption isolates the acquisition-timing problem, while DSA estimates the corresponding quantities online. The appendix gives the full causal model, independence conditions, admissible policy class, and proofs.

5.2 Exact Finite-Horizon Optimal Policy

Consider an inactive type j when n periods remain, including the current period. If the type is built now, its expected current contribution is $\rho_j - b_j$. With probability q_j, admission additionally creates an expected $(n-1)p_j$ future reuse opportunities, each worth s_j. This motivates the acquisition score

$$\Psi_j(n) := \rho_j + q_j(n-1)p_j s_j - b_j. \tag{2}$$

The score resembles a one-shot net-present-value calculation. The nontrivial point is that its sign remains decision-equivalent to the full dynamic program even though the agent may wait and retry failed acquisitions.

Theorem 1 (Exact finite-horizon optimal acquisition)**.** *Under Assumption 1, the following policy minimizes expected cumulative cost among all causal patch/build/reuse policies in the stated model. Reuse an admitted type-j tool whenever type j arrives. If type j is inactive with n periods remaining, build if $\Psi_j(n) > 0$, patch if $\Psi_j(n) < 0$, and either action is optimal at equality.*

For the strict-positive convention used in our experiments, when $b_j \ge \rho_j$ the smallest remaining horizon at which building becomes strictly optimal is

$$n_j^+ = 2 + \left\lfloor \frac{b_j - \rho_j}{q_j p_j s_j} \right\rfloor, \tag{3}$$

while $n_j^+ = 1$ when $b_j < \rho_j$. Acquisition therefore becomes more attractive with higher recurrence p_j, admission probability q_j, or reuse saving s_j, and less attractive with larger build cost b_j or a shorter remaining horizon.

Why the one-line score is dynamically optimal. Let $U_j(n) = np_j s_j$ be the expected future saving when type j is already active, and let $V_j(n)$ be the optimal expected net saving when it is inactive before the next arrival is observed. At a type-j arrival, the true Bellman advantage of building over patching is

$$A_j(n) = \rho_j - b_j + q_j\{U_j(n-1) - V_j(n-1)\} = \Psi_j(n) - q_j V_j(n-1). \tag{4}$$

A Bellman induction shows that $\operatorname{sign}(A_j(n)) = \operatorname{sign}(\Psi_j(n))$ for every n. Therefore the simple score identifies the exact optimal action even though it does not generally equal the magnitude of the dynamic advantage. The appendix gives the complete induction, tie cases, and a closed form for $V_j(n)$. Within this known-parameter model, waiting alone cannot make an unattractive acquisition become attractive because $\Psi_j(n)$ decreases as the remaining horizon shrinks.

By contrast, if a profitable build fails admission, retry can remain optimal while the remaining horizon stays above the threshold. When demand is unknown, however, future arrivals can reveal information; Proposition 5 gives an example in which that information creates value of waiting.

5.3 Exact Gap of Build First

In the separable model, define Build First to make exactly one acquisition attempt at the first occurrence of each inactive type, reuse after admission, and never retry after non-admission. Its gap to the dynamic optimum can be characterized exactly.

Let π^{BF} denote this type-level Build First policy and π^* the optimal policy from Theorem 1. Conditional on the first type-j arrival occurring with n periods remaining, define

$$\ell_j^{\mathrm{BF}}(n) := \begin{cases} -\Psi_j(n), & \Psi_j(n) < 0, \\ 0, & \Psi_j(n) = 0, \\ (1-q_j)V_j(n-1), & \Psi_j(n) > 0. \end{cases} \tag{5}$$

Theorem 2 (Exact Build-First cost-gap decomposition)**.** *Under Assumption 1, the exact expected cost gap between Build First and the known-parameter optimal causal policy is*

$$J_T(\pi^{\mathrm{BF}}) - J_T(\pi^*) = \sum_j \sum_{u=1}^{T} p_j(1-p_j)^{u-1} \ell_j^{\mathrm{BF}}(T-u+1). \tag{6}$$

The gap consists of exactly two nonnegative mechanisms: premature acquisition*, when Build First acquires at a first occurrence with* $\Psi_j(n) < 0$*, and* failure to retry*, when* $\Psi_j(n) > 0$ *but non-admission causes Build First to discard the positive continuation value* $V_j(n-1)$*.*

Interpretation. The first term formalizes the cost of treating every newly observed type as equally worthy of investment: low-recurrence or late-arriving types can have insufficient remaining demand to repay acquisition. The second term captures the opposite rigidity: rejection of a worthwhile candidate does not imply that the capability itself has become uneconomic. Here "cost gap" is relative to the known-parameter causal optimum, not a hindsight policy that observes future arrivals or generation outcomes.

The proof conditions on the first arrival time and uses the monotonicity of $\Psi_j(n)$ as the horizon shrinks. If $\Psi_j(n) < 0$, the optimal policy patches and never builds that type later, so Build First loses exactly $-\Psi_j(n)$. If $\Psi_j(n) > 0$, both policies build now, but after non-admission Build First gives up while the optimum retains continuation value $V_j(n-1)$, producing gap $(1-q_j)V_j(n-1)$. Averaging these conditional gaps over the geometric first-arrival distribution gives Equation (6); the appendix provides the full proof.

5.4 Connection to the Executable Controller

There is an exact correspondence in a cell-separable specialization of the benchmark. If each family–pattern service cell is treated as one type, cross-pattern transfer is removed, parameters are known, and generation trials are fresh, then the cell is inactive exactly while the acquisition gate is open. Pattern-First therefore coincides with the type-level Build First policy in Theorem 2: it attempts one build at the first eligible occurrence and never retries after non-admission.

The executable benchmark is richer: admitted transformations can have latent cross-pattern transfer, one admission changes future gate states, the candidate pool is finite, and DSA learns demand and quality online. Family-First is a coarser family-level analogue. The theorem therefore supplies the structural acquisition principle and two model-level failure modes, while measured policy differences also reflect these operational effects. Under parameter uncertainty, DSA uses posterior estimates of the same acquisition structure; Proposition 5 gives an example in which future arrivals create additional information value.

6 Experiments

This section tests DSA with executable program transformations under controlled recurrence profiles and a broader probabilistic demand sweep.

Executable benchmark. We construct 12 API-migration families: four function renames, four keyword renames, and four constructor migrations. Each family appears through three observable coding patterns (direct use, symbol aliasing, and module aliasing). For every family–pattern pair, we generate three one-shot candidate LibCST transformations with Codex using model `gpt-5.6-terra`, giving 108 candidates in total. The pool is frozen before online evaluation. Of the 108 candidates, 99 (91.7%) solve the triggering client but only 77 (71.3%) pass the fixed admission test for persistent reuse, directly instantiating the G/I distinction in Definition 1. Patch is modeled as a reliable one-off service and remains available after unsuccessful build or reuse attempts, so all evaluated trajectories finish with the same task success; we compare *modeled cost*, not dollars or wall-clock latency.

Baselines. In addition to family-level *Family-First*, which makes one designated attempt when an API family first appears, we evaluate *Pattern-First*: it makes one acquisition attempt at the first gate-eligible occurrence of each family–pattern pair and never retries that pair. Pattern-First is a stronger first-seen comparator because it can expand operational pattern coverage.

(a) Policy summary

Policy	Cost	DSA − comparator [95% CI]	Builds
DSA (Ours)	**583.90**	—	18.96
Pattern-First	625.79	-41.88 $[-47.05, -36.71]$	22.00
Always-Build-When-Eligible	630.24	-46.34 $[-51.14, -41.53]$	28.00
Family-First + Retry	719.65	-135.75 $[-142.87, -128.63]$	14.69
Family-First	730.33	-146.43 $[-153.05, -139.81]$	12.00

(b) Paired costs

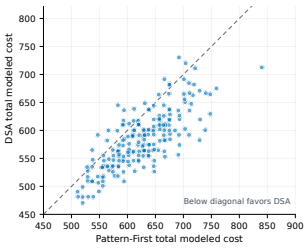


Figure 3: **Heavy-tailed fixed-count comparison.** (a) Per-stream mean modeled cost and build count over 200 paired streams; differences are DSA minus comparator, with paired 95% Student-t intervals. (b) Paired total modeled costs for DSA and Pattern-First; points below the diagonal favor DSA. DSA is cheaper on 178 of 200 streams and has 6.69% lower mean cost. Intervals condition on the single frozen candidate population and quantify stream-to-stream variability.

Additional baselines include Family-First with retry, Always-Build-When-Eligible, and Always Patch; all use the same frozen candidate population, scope semantics, costs, validation, and fallback rules. Under the benchmark gate, Always-Build-When-Eligible is also the operational counterpart of adding unconditional retry to Pattern-First.

Heavy-tailed controlled profile. With DSA and all baselines fixed in advance, we evaluate 200 previously unused heavy-tailed task-stream seeds over the same frozen candidate population. DSA has mean modeled cost 583.90 versus 625.79 for Pattern-First, a paired difference of -41.88 (95% CI $[-47.05, -36.71]$), corresponding to 6.69% lower mean cost; DSA is cheaper on 178/200 paired streams. DSA also improves over Always-Build-When-Eligible and retains the larger advantage over family-level first-seen acquisition. Figure 3 summarizes the policy comparison and the paired DSA–Pattern-First costs.

Operational behavior. Under heavy-tailed demand, DSA makes fewer builds than Pattern-First (18.96 versus 22.00) while obtaining similar successful reuse (41.53 versus 40.64); the aggregate successful-reuse/admission ratio (i.e. successful reuses per admitted tool) is 2.91 versus 2.46. Removing demand adaptation raises DSA's heavy-tail cost by 24.80 units, and disabling retry raises it by 14.16, with both paired intervals excluding zero. In the family-balanced fixed-count control, Pattern-First is instead cheaper than DSA (612.46 versus 640.03; paired difference $+27.57$, 95% CI $[24.15, 30.98]$), and both component removals reduce DSA cost there. These fixed-count profiles deliberately control realized family counts and therefore do not match DSA's i.i.d. categorical demand model; a separate i.i.d. Zipf sweep removes this mismatch while realized recurrence remains random. In that sweep, DSA has lower mean cost than Pattern-First in all 63 tested cells, with 61/63 remaining resolved after Holm correction. The appendix further reports build-cost sensitivity and targeted diagnostics of the cross-pattern value proxy and hierarchical quality initialization; at sufficiently high build cost, never building can be preferable to either acquisition policy.

7 Conclusion

We study when an agent should invest in building a persistent reusable capability rather than patch the current task or reuse what it already has. In a separable stochastic-admission model, we derive an exact finite-horizon optimal acquisition policy and an exact decomposition of the type-level Build-First cost gap into premature acquisition and failure to retry. In executable API migration, DSA lowers modeled cost relative to stronger first-seen acquisition under heavy-tailed recurrence, while the value of selective acquisition depends on the recurrence structure and cost regime.

Potential extensions. API migration provides a controlled setting with explicit recurrence, validation, and reuse, while broader coding, data-analysis, and web-agent tasks may require learning task similarity and transfer scope. More autonomous systems may also refine, compose, replace, or retire tools as demand and environments evolve. Section C.2 discusses these extensions together with changing demand, online generation, and richer resource objectives.

AI USE STATEMENT

In this work, we used generative AI tools as research assistants for literature search and synthesis; brainstorming and refinement of theoretical ideas and hypotheses; mathematical derivations and proof development; experimental-design feedback; code implementation, debugging, and data processing; result analysis and interpretation; figure preparation; and drafting and editing portions of the manuscript. Separately, LLM-generated program transformations are part of the experimental system described in Section 6. The authors led the research ideas, problem formulation, methodology and experiment design, scientific analysis, writing, and presentation; all AI-assisted mathematical claims, code, figures, and text were reviewed and revised by the authors as needed. We take responsibility for the final content of this paper.

REPRODUCIBILITY STATEMENT

The formal stochastic model, notation, and complete proofs are provided in Appendix A. The controller specification, normalized cost accounting, validation rules, stream generators, baseline definitions, candidate-pool audit, seed ranges, and extended numerical results are provided in Appendix B. The reported policy comparisons use frozen candidate files and stream manifests, with paired seed-level inference. The accompanying anonymized supplementary artifact contains the frozen candidate pool and integrity metadata, benchmark clients and tests, policy implementations, stream manifests, and run-level outputs needed to reproduce the reported tables and figures.

REFERENCES

Noga Alon, Baruch Awerbuch, and Yossi Azar. The online set cover problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pp. 100–105, 2003.

Ashwinkumar Badanidiyuru, Robert Kleinberg, and Aleksandrs Slivkins. Bandits with knapsacks. In *Annual Symposium on Foundations of Computer Science (FOCS-13)*, pp. 207–216. IEEE, 2013.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, volume 2024, pp. 54067–54089, 2024.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, et al. Adaevolve: Adaptive LLM driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

Keyuan Cheng, Xudong Shen, Yihao Yang, Tengyue Wang, Yang Cao, Muhammad Asif Ali, Hanbin Wang, Lijie Hu, and Di Wang. Codemenv: Benchmarking large language models on code migration. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 2719–2744, 2025.

Christian Coester and Alex Turoczy. Primal-dual online algorithms for the parking permit problem. *arXiv preprint arXiv:2607.08262*, 2026.

Tristan Coignion, Clément Quinton, and Romain Rouvoy. When faster isn't greener: The hidden costs of llm-based code optimization. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1655–1666. IEEE, 2025.

Qiming Cui and Michael Dinitz. Ski rental with distributional predictions of unknown quality. In *International Conference on Machine Learning*, 2026.

Chris Cummins, Volker Seeker, Jordi Armengol-EstapÃŠ, Aram H Markosyan, Gabriel Synnaeve, and Hugh Leather. Don't transform the code, code the transforms: Towards precise code rewriting using LLMs. *arXiv preprint arXiv:2410.08806*, 2024.

Zijie Dai, Shiyuan Deng, Sheng Guan, Yizhou Tian, Xin Yao, Xiao Yan, and James Cheng. RecMem: Recurrence-based memory consolidation for efficient and effective long-running LLM agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 32353–32376, 2026.

Ilias Diakonikolas, Vasilis Kontonis, Christos Tzamos, Ali Vakilian, and Nikos Zarifis. Learning online algorithms with distributional advice. In *International Conference on Machine Learning*, pp. 2687–2696. PMLR, 2021.

Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. Unprecedented code change automation: The fusion of LLMs and transformation by example. *Proceedings of the ACM on Software Engineering*, 1(FSE):631–653, 2024.

Yicheng Feng, Yan Zhang, Yan Cheng, and Wei Qi. Scores are not decisions: Cost-aware stopping for tool acquisition in llm agents. *arXiv preprint arXiv:2607.27083*, 2026.

Ganesh Ghalme, Swapnil Dhamal, Shweta Jain, Sujit Gujar, and Y. Narahari. Ballooning multi-armed bandits. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems*, pp. 1849–1851, 2020.

Mohayeminul Islam, Ajay Kumar Jha, Sarah Nadi, and Ildar Akhmetov. Pymigbench: A benchmark for python library migration. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 511–515. IEEE, 2023.

Anand Kalvit and Assaf Zeevi. Bandits with dynamic arm-acquisition costs. In *2022 58th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1–7. IEEE, 2022.

Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. CODESKILL: Learning self-evolving skills for coding agents. *arXiv preprint arXiv:2605.25430*, 2026.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

Zvi Lotker, Boaz Patt-Shamir, and Dror Rawitz. Rent, lease or buy: Randomized algorithms for multislope ski rental. *arXiv preprint arXiv:0802.2832*, 2008.

Aleksander Madry and Debmalya Panigrahi. The semi-stochastic ski-rental problem. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

Robert McDonald and Daniel Siegel. The value of waiting to invest. *The quarterly journal of economics*, 101(4):707–727, 1986.

Adam Meyerson. Online facility location. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pp. 426–431. IEEE, 2001.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Huiyun Peng, Arjun Gupte, Ryan Hasler, Nicholas John Eliopoulos, Chien-Chou Ho, Rishi Mantri, Leo Deng, Konstantin Läufer, George K Thiruvathukal, and James C Davis. Sysllmatic: Large language models are software system optimizers. *Journal of Systems and Software*, pp. 112929, 2026.

Robert S Pindyck. Investments of uncertain cost. *Journal of financial Economics*, 34(1):53–76, 1993.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, 2023.

Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. MELT: mining effective lightweight transformations from pull requests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1516–1528. IEEE, 2023.

Daniel Ramos, Catarina Gamboa, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. SPELL: Synthesis of programmatic edits using LLMs. *arXiv preprint arXiv:2602.01107*, 2026.

Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. BUMP: A benchmark of reproducible breaking dependency updates. In *2024 IEEE International conference on software analysis, evolution and reengineering (SANER)*, pp. 159–170. IEEE, 2024.

Frank Reyes, Benoit Baudry, and Martin Monperrus. Agentic generation of AST transformation rules for fixing breaking updates. *arXiv preprint arXiv:2606.24446*, 2026a.

Frank Reyes, May Mahmoud, Federico Bono, Sarah Nadi, Benoit Baudry, and Martin Monperrus. Byam: Fixing breaking dependency updates with large language models. *Empirical Software Engineering*, 31(4):113, 2026b.

Xintian Shen, Jiawei Chen, Lihao Zheng, Hao Ma, Tao Wei, and Kun Zhan. Evolving from tool user to creator via training-free experience reuse in multimodal reasoning. *arXiv preprint arXiv:2602.01983*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Yangbo Wei, Zhen Huang, Shaoqiang Lu, Junhong Qian, Qifan Wang, Chen Wu, and Lei He. SkillSmith: Co-evolving skills and tools for self-improving agent systems. *arXiv preprint arXiv:2606.01314*, 2026.

Binghan Wu, Wei Bao, Dong Yuan, and Bing Zhou. Competitive analysis for multi-commodity ski-rental problem. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2022.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. SkillRL: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

Duling Xu, Zheng Chen, Zaifeng Pan, Jiawei Guan, Dong Dong, Jialin Li, and Bangzheng Pu. SkillSmith: Compiling agent skills into boundary-guided runtime interfaces. *arXiv preprint arXiv:2605.15215*, 2026a.

Jianyu Xu, Vidhi Jain, Bryan Wilder, and Aarti Singh. Online decision making with generative action sets. In *International Conference on Learning Representations*, 2026b.

Xiaonan Xu and Wenjing Wu. Compression, structure, and executor capability: a controlled real-cost decomposition of language-model agent skill optimisation. *arXiv preprint arXiv:2607.03048*, 2026.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *International Conference on Learning Representations*, volume 2024, pp. 40097–40125, 2024.

Murong Yue, Zhiwei Liu, Liangwei Yang, Jianguo Zhang, Zuxin Liu, Haolin Chen, Ziyu Yao, Silvio Savarese, Caiming Xiong, Shelby Heinecke, et al. Toollibgen: Scalable automatic tool creation and aggregation for LLM reasoning. *arXiv preprint arXiv:2510.07768*, 2025.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Appendix

Appendix Roadmap. Appendix A gives the formal model, notation, and complete proofs. Appendix B specifies the executable benchmark and controller and reports extended numerical results. Appendix C provides extended related work and broader limitations and extensions.

A Formal Model and Proofs

This appendix makes the separable model fully explicit, proves the two theorem-level claims in Section 5, and records auxiliary results about uncertainty, overlap, and numerical integration. The exact dynamic guarantees apply to the known-parameter separable model; DSA uses these quantities through posterior estimates in the richer executable benchmark.

A.1 Notation and Benchmark Mapping

Table 1: **Notation used across the main paper and appendix.**

Symbol	Meaning
X_t	Generic task type arriving at time t.
f_t, r_t	Exact API family and observable coding pattern in the executable benchmark.
z_t	Observable build context derived from the arriving task and its features.
$\mathcal{L}_t$	Library of admitted tools available before the time-t decision.
G_z, I_z	Current-task success and durable admission of a new candidate; $I_z \le G_z$.
σ_z, R_z	Patch-relative saving conditional on current success, and realized current saving $R_z = G_z\sigma_z$.
b_z	Build and validation cost in context z.
A	Prospective candidate tool in the DSA score.
$N_k, S_{A,k}$	Predicted remaining count and per-use saving for benchmark service cell (f,k); k indexes the three patterns of the current family.
$\widehat{\Delta}_t$	DSA posterior-mean acquisition score.
j	Abstract separable capability type in the exact theory.
$p_j, q_j, s_j, \rho_j, b_j$	Arrival probability, admission probability, persistent per-use saving, mean current saving $\rho_j = \mathbb{E}[R_j]$, and build cost in the separable model.
$\Psi_j(n)$	Closed-form separable acquisition score with n periods remaining.
$A_j(n)$	True Bellman advantage of build over patch in the separable model.

A family–pattern service cell becomes an exact separable type in the idealized specialization with no cross-pattern transfer, fresh independent generation, and known parameters; under that specialization Pattern-First is the type-level Build First policy. The executable benchmark retains transfer, operational scope, online learning, and a finite shared candidate pool, so its cells need not evolve independently. Section 5.4 uses this specialization only as the theory–benchmark correspondence.

A.2 Formal Separable Model

Fix a finite horizon T and type set $[K]$. Before the time-t action, the agent observes the current type X_t. The arrival process is

$$X_1, \dots, X_T \overset{\text{i.i.d.}}{\sim} \text{Categorical}(p), \qquad p_j > 0, \qquad \sum_{j=1}^{K} p_j = 1.$$

Let $\mathcal{F}_{t-1}$ be the sigma-field generated by earlier arrivals, actions, observed service outcomes, and observed build outcomes, and let $\mathcal{H}_t = \sigma(\mathcal{F}_{t-1}, X_t)$ be the information available for the current decision. A causal policy chooses an action measurable with respect to $\mathcal{H}_t$. At most one build may be attempted in a period, and a build can target only the currently arriving type.

For each type j, patching defines the zero-saving reference. If type j is inactive, a build costs $b_j > 0$ and yields a pair (R_j, I_j), where R_j is the current patch-relative saving and $I_j \in \{0,1\}$ indicates durable admission. We allow dependence between current success and admission within the same

build; in particular, $I_j \le G_j$ in the underlying current-success representation and $\rho_j := \mathbb{E}[R_j]$. Conditional on the fixed model parameters, build outcomes are i.i.d. across attempts of the same type and independent of future task arrivals. Non-admission leaves the type inactive and permits a fresh later attempt.

After admission, type j remains active for the rest of the horizon and yields known net expected saving $s_j > 0$ on every future type-j arrival. This saving already includes any expected application and detected-failure fallback cost. The admitted capability has no value on other types. There is no discounting, holding cost, acquisition budget, capacity coupling, or cross-type composition. We impose $\rho_j - b_j \le s_j$ only to ensure that rebuilding an already active identical capability cannot improve on reuse. Since $p_j > 0$ and $q_j := \mathbb{P}(I_j = 1) > 0$, the proof below uses $0 < p_j q_j \le 1$.

For a policy π, let $J_T(\pi)$ denote expected cumulative modeled cost. Let C_t^P be the patch-only cost on the realized task and let Z_t be the realized service saving relative to patching, excluding acquisition cost. Pathwise,

$$\sum_{t=1}^{T} \left(C_t^{\text{service}} + b_{X_t} \mathbf{1}\{\text{build at } t\} \right) = \sum_{t=1}^{T} C_t^P - \sum_{t=1}^{T} \left(Z_t - b_{X_t} \mathbf{1}\{\text{build at } t\} \right).$$

The first term is policy independent because arrivals are exogenous. Hence minimizing J_T is equivalent to maximizing expected service saving net of build cost. All value functions below use this net-saving representation.

A.3 Joint Bellman Decomposition

Define $V_j(0) = 0$ recursively by

$$V_j(n) = V_j(n-1) + p_j \left[\rho_j - b_j + q_j \{ U_j(n-1) - V_j(n-1) \} \right]_+, \qquad U_j(n) := n p_j s_j.$$

We now show that these one-coordinate recursions are exactly the joint optimal value functions.

Lemma 3 (Type decomposition and Bellman recursion)**.** *Let $W_n(S)$ be the optimal expected net saving over the next n periods before the next arrival is observed, when $S \subseteq [K]$ is the set of active types. Under Assumption 1,*

$$W_n(S) = \sum_{j \in S} U_j(n) + \sum_{j \notin S} V_j(n).$$

At an inactive type-j arrival with n periods remaining, the build-versus-patch Bellman advantage is

$$A_j(n) = \rho_j - b_j + q_j \{ U_j(n-1) - V_j(n-1) \}.$$

Proof. The identity is immediate at $n = 0$. Assume it holds for horizon $n-1$ and fix an active set S.

If the arriving type j is already in S, reuse is optimal: rebuilding creates no persistent increment, while $\rho_j - b_j \le s_j$ makes its expected current net contribution no larger than reuse. Since $s_j > 0$, reuse also strictly dominates patch. The current saving is s_j and the continuation state remains S.

Now suppose $j \notin S$. Patching gives continuation $W_{n-1}(S)$. Building gives current net saving $\rho_j - b_j$ and continuation

$$q_j W_{n-1}(S \cup \{j\}) + (1 - q_j) W_{n-1}(S).$$

By the induction hypothesis, every coordinate other than j is identical in the two continuations, and

$$W_{n-1}(S \cup \{j\}) - W_{n-1}(S) = U_j(n-1) - V_j(n-1).$$

Therefore the conditional build advantage is exactly $A_j(n)$. Averaging over the categorical arrival gives

$$W_n(S) = W_{n-1}(S) + \sum_{j \in S} p_j s_j + \sum_{j \notin S} p_j [A_j(n)]_+.$$

Applying the induction hypothesis coordinatewise yields the stated recursions for U_j and V_j, proving the decomposition at horizon n. □

A.4 Proof of the Exact Finite-Horizon Policy

Proof of Theorem 1. Fix a type j and suppress its index, writing p, q, s, b, ρ. Let $D(n) = U(n) - V(n)$. By Lemma 3, building at an inactive arrival is optimal exactly when

$$D(n-1) \geq \theta, \qquad \theta := \frac{b-\rho}{q}.$$

The recursion for D is

$$D(n) = D(n-1) + ps - p[\rho - b + qD(n-1)]_+.$$

First suppose $b > \rho$, so $\theta > 0$. Before the first crossing of θ, the positive-part term is zero. Starting from $D(0) = 0$, induction therefore gives $V(n) = 0$ and $D(n) = nps$ throughout this region. Consequently, at a decision with n periods remaining the first weakly profitable comparison is

$$(n-1)ps \geq \frac{b-\rho}{q},$$

which is equivalent to

$$\Psi(n) = \rho + q(n-1)ps - b \geq 0.$$

It remains to show that the build region is invariant after the first crossing. Whenever $D(n-1) \geq \theta$,

$$D(n) = (1-pq)D(n-1) + pqD^*, \qquad D^* := \frac{s-\rho+b}{q} = \theta + \frac{s}{q}.$$

Because $0 < pq \leq 1$ and $D^* > \theta$, every value at or above θ is mapped strictly above θ. Hence the Bellman advantage and $\Psi(n)$ have the same sign at every horizon, including equality at the first weak crossing.

If $b \leq \rho$, then $\theta \leq 0$ and $D(0) = 0 \geq \theta$, so the same invariant-region argument begins at $n = 1$. The last-period decision is a tie when $b = \rho$ and strictly favors building when $b < \rho$.

Solving $\Psi_j(n) > 0$ over integer n gives

$$n_j^+ = \begin{cases} 1, & b_j < \rho_j, \\ 2 + \left\lfloor \dfrac{b_j - \rho_j}{q_j p_j s_j} \right\rfloor, & b_j \geq \rho_j. \end{cases}$$

By Lemma 3, the joint value decomposes across coordinates, so choosing the Bellman-optimal action for the arriving coordinate at every history is globally optimal among all causal policies in the stated model. This proves Theorem 1. □

The same recursion gives a useful closed form. Let n_0 be the smallest $n \geq 1$ with $\Psi(n) \geq 0$, put $d_0 = (n_0 - 1)ps$, $\lambda = 1 - pq$, and $D^* = (s - \rho + b)/q$. Then $V(n) = 0$ for $n < n_0$, and for $n \geq n_0 - 1$,

$$D(n) = D^* - (D^* - d_0)\lambda^{n-n_0+1}, \qquad V(n) = nps - D(n).$$

For $n \geq n_0$,

$$A(n) = s - q(D^* - d_0)\lambda^{n-n_0}.$$

In particular, in the build region $0 \leq A(n) \leq s$ and $A(n) \to s$ as the horizon grows. Moreover,

$$A_j(n) = \Psi_j(n) - q_j V_j(n-1),$$

so Ψ_j is sign-equivalent to the Bellman advantage without generally being equal to its magnitude.

A.5 Proof of the Build-First Cost Gap

Proof of Theorem 2. Fix type j and let τ_j be its first arrival time. For $1 \leq u \leq T$,

$$\mathbb{P}(\tau_j = u) = p_j(1-p_j)^{u-1}.$$

Condition on $\tau_j = u$ and write $n = T - u + 1$. Before this arrival, the type is inactive under both policies. Because the net-saving objective is additive across types and Build First's type-j actions

depend only on the type-j history, its expected value decomposes across types just as the optimal value does; it therefore suffices to compute the conditional gap for a fixed j.

A one-shot Build-First attempt has expected current-plus-future net saving

$$\rho_j - b_j + q_j U_j(n-1) = \rho_j - b_j + q_j(n-1)p_j s_j = \Psi_j(n).$$

This equality uses only the mean current saving ρ_j and marginal admission probability q_j; it does not require independence between current success and admission within an attempt.

If $\Psi_j(n) < 0$, Theorem 1 says the optimal policy patches. Since the remaining horizon only decreases, every later score is weakly smaller, so the optimal policy never builds this type later. Its conditional net saving is zero, whereas Build First obtains $\Psi_j(n) < 0$, producing cost gap $-\Psi_j(n)$.

If $\Psi_j(n) = 0$, patch and build are both optimal and the shorter continuation has no positive acquisition value. The conditional gap is zero.

If $\Psi_j(n) > 0$, both policies build now. The optimal conditional value is

$$\rho_j - b_j + q_j U_j(n-1) + (1-q_j)V_j(n-1),$$

whereas Build First receives the same current payoff and admitted continuation but gives up after non-admission. Its value is only $\Psi_j(n)$. The exact conditional gap is therefore $(1-q_j)V_j(n-1)$.

These three cases give $\ell_j^{\mathrm{BF}}(n)$. Averaging over the first-arrival distribution and summing over types yields

$$J_T(\pi^{\mathrm{BF}}) - J_T(\pi^*) = \sum_j \sum_{u=1}^{T} p_j(1-p_j)^{u-1}\ell_j^{\mathrm{BF}}(T-u+1),$$

which proves Theorem 2. Equivalently, the two nonnegative components are

$$\mathcal{R}_T^{\mathrm{early}} = \sum_j \sum_{u=1}^{T} p_j(1-p_j)^{u-1}[-\Psi_j(T-u+1)]_+,$$

and

$$\mathcal{R}_T^{\mathrm{retry}} = \sum_j \sum_{u=1}^{T} p_j(1-p_j)^{u-1}\mathbf{1}\{\Psi_j(T-u+1) > 0\}(1-q_j)V_j(T-u).$$

Their sum is the Build-First cost gap to the known-parameter causal optimum. □

A.6 Decision Stability under Parameter Error

The exact policy assumes known parameters, whereas DSA estimates demand and generation quality. The following deterministic result identifies when estimation error can change a separable acquisition decision. Treat b as known and define

$$\widehat{\Psi}(n) = \widehat{\rho} + (n-1)\widehat{p}\widehat{q}\widehat{s} - b.$$

Proposition 4 (Deterministic score stability)**.** *Suppose*

$$|\widehat{\rho} - \rho| \le \epsilon_\rho, \qquad \left|\frac{\widehat{p}}{p} - 1\right| \le \epsilon_p, \qquad \left|\frac{\widehat{q}}{q} - 1\right| \le \epsilon_q, \qquad \left|\frac{\widehat{s}}{s} - 1\right| \le \epsilon_s.$$

Then

$$|\widehat{\Psi}(n) - \Psi(n)| \le \epsilon_\rho + (n-1)pqs\,[(1+\epsilon_p)(1+\epsilon_q)(1+\epsilon_s) - 1] =: E_n.$$

If $|\Psi(n)| > E_n$, the estimated and true strict-positive decisions agree. Any disagreement away from a true tie requires $|\Psi(n)| \le E_n$.

Proof. Write $\widehat{p} = p(1+e_p)$ and similarly for q and s. Expanding the product and applying the triangle inequality gives

$$|(1+e_p)(1+e_q)(1+e_s) - 1| \le (1+\epsilon_p)(1+\epsilon_q)(1+\epsilon_s) - 1.$$

Multiplying by $(n-1)pqs$ and adding the current-saving error proves the bound. If the two signs differ away from a tie, zero lies between the two scores, so $|\Psi(n)| \le |\widehat{\Psi}(n) - \Psi(n)|$. □

This proposition concerns the idealized separable score Ψ; it is not a stability guarantee for the full Monte Carlo controller in Algorithm 1, whose working score also includes benchmark-specific demand, transfer, and scope approximations.

A.7 Waiting under Demand Uncertainty

The known-model monotonicity statement in Section 5.2 does not eliminate option value when future arrivals reveal information. The following example isolates that distinction.

Proposition 5 (Free demand information can make waiting optimal)**.** *Even with perfect type-specific tools and no overlap, a positive posterior expected one-acquisition score need not make immediate building Bayes-optimal when demand is unknown and future arrivals provide information.*

Proof. Consider two task types and four periods. Type 1 can be built at cost $3/2$, succeeds with certainty, and saves one on the current and every later type-1 task; type 2 is always patched. The type-1 arrival probability is either 1 or $1/100$, with prior probability $1/401$ on the first case. Conditional on observing type 1 in period 1, Bayes' rule gives posterior probabilities $1/5$ and $4/5$ for the two demand states. Therefore

$$\mathbb{E}[p] = \frac{26}{125}, \qquad \mathbb{E}[p^2] = \frac{2501}{12500}.$$

Building immediately has posterior expected net saving

$$1 + 3\mathbb{E}[p] - \frac{3}{2} = \frac{31}{250} = 0.124 > 0.$$

Now consider the feasible policy that patches in period 1, builds in period 2 only if that task is again type 1, and otherwise never builds. Its conditional expected saving is

$$\mathbb{E}\left[p\left(1 + 2p - \frac{3}{2}\right)\right] = 2\mathbb{E}[p^2] - \frac{1}{2}\mathbb{E}[p] = \frac{1851}{6250} = 0.29616.$$

The waiting policy is strictly better even though immediate posterior-mean lifetime value is positive. It uses only the next observed arrival, not knowledge of the latent demand parameter. □

A.8 Asymmetry of Over- and Under-Building

For this subsection consider policies that always reuse an active type and choose only between patch and build when a type is inactive. Let $\Gamma_t(\pi)$ be the Bellman action gap at the state visited by π at time t: the optimal net-saving value minus the value of the action selected by π. The finite-horizon performance-difference identity is

$$\mathcal{R}_T(\pi) := J_T(\pi) - J_T(\pi^*) = \mathbb{E}_\pi\left[\sum_{t=1}^{T} \Gamma_t(\pi)\right].$$

Define $\mathcal{R}_T^{\text{over}}(\pi)$ as the contribution from inactive states where π builds while the oracle strictly patches, and $\mathcal{R}_T^{\text{under}}(\pi)$ as the contribution from inactive states where π patches while the oracle strictly builds. At such a state the one-step gap is $|A_j(n)|$.

Proposition 6 (Bounded over-building and potentially linear under-building)**.** *For every history-dependent policy in the class above,*

$$\mathcal{R}_T^{\text{over}}(\pi) \le \sum_{j=1}^{K} \frac{(b_j - \rho_j)_+}{q_j},$$

which is independent of T for fixed K. In contrast, there is a one-type instance for which a never-build policy incurs under-building gap $T - 1$.

Proof. When the oracle strictly patches type j, the Bellman advantage equals the negative score because the continuation acquisition value is zero. An unnecessary build therefore loses

$$-A_j(n) = b_j - \rho_j - q_j(n-1)p_j s_j \le (b_j - \rho_j)_+.$$

If $b_j > \rho_j$, then $(b_j - \rho_j)/q_j$ is exactly the threshold level θ_j in the proof of Theorem 1. Each mistaken build is a fresh admission attempt. Once an attempt is admitted, the type becomes active and the policy class reuses it thereafter, so no later inactive-state over-building mistake is possible for that type. Conditional on the policy's chosen mistaken opportunities, the number of attempts before

admission is stochastically dominated by a geometric random variable with mean $1/q_j$. Multiplying the per-attempt bound by $1/q_j$ and summing over types proves the first claim. Thus over-building is self-limiting: eventual admission closes the inactive acquisition opportunity.

For the contrast, take $K = 1$ with $p = q = s = 1$ and $\rho = b = 1$. Then $\Psi(n) = n - 1$, so the oracle builds for every $n \geq 2$ and is indifferent at $n = 1$. The Bellman advantage is one throughout the strict build region. A never-build policy therefore loses one on each of the first $T-1$ periods and zero at the final tie. Under-building need not self-correct because patching leaves the type inactive. □

A.9 Frozen-Library Marginal Value and Overlap

The deployed DSA score uses a hard operational scope gate and does not subtract an additional future-overlap term. The following fixed-library identity is included to interpret the scope and overlap diagnostics in Section B; it is not an optimality statement for DSA.

Fix nonnegative demand weights w_k and nonnegative per-use savings $s_{a,k}$ after applying the operational scope mask. For library $\mathcal{L}$, define

$$F_w(\mathcal{L}) = \sum_{k=1}^{K} w_k m_k(\mathcal{L}), \qquad m_k(\mathcal{L}) = \max\left(\{0\} \cup \{s_{a,k} : a \in \mathcal{L}\}\right).$$

Proposition 7 (Marginal value and overlap bias)**.** *For any candidate a,*

$$F_w(\mathcal{L} \cup \{a\}) - F_w(\mathcal{L}) = \sum_k w_k [s_{a,k} - m_k(\mathcal{L})]_+.$$

Standalone value exceeds this frozen-library marginal value by exactly

$$\sum_k w_k s_{a,k} - \{F_w(\mathcal{L} \cup \{a\}) - F_w(\mathcal{L})\} = \sum_k w_k \min\{s_{a,k}, m_k(\mathcal{L})\}.$$

Consequently F_w is monotone and submodular.

Proof. For each coordinate,

$$m_k(\mathcal{L} \cup \{a\}) - m_k(\mathcal{L}) = [s_{a,k} - m_k(\mathcal{L})]_+.$$

Summing with nonnegative weights gives the first identity. The second follows from $x - [x-y]_+ = \min\{x, y\}$ for $x, y \geq 0$. If $\mathcal{L}_1 \subseteq \mathcal{L}_2$, then $m_k(\mathcal{L}_1) \leq m_k(\mathcal{L}_2)$, so every coordinate marginal weakly decreases as the library grows, establishing diminishing returns and hence submodularity. □

A tool that works physically on a pattern but does not declare that pattern is absent from the operational baseline $m_k(\mathcal{L})$. Physical redundancy can therefore coexist with positive operational value, which is why hidden coverage is kept separate from selectable declared coverage in the executable analysis.

A.10 Finite Monte Carlo Approximation

At a fixed observable DSA state, let $Z_1, \ldots, Z_M$ be independent score draws under the stored working posterior and let $B = \mathbb{E}[Z_1]$. Suppose all draws lie in an interval of deterministic width W.

Proposition 8 (Fixed-sample numerical sign bound)**.** *For any $\epsilon > 0$,*

$$\mathbb{P}\left(\left|\frac{1}{M}\sum_{m=1}^{M} Z_m - B\right| \geq \epsilon\right) \leq 2\exp\left(-\frac{2M\epsilon^2}{W^2}\right).$$

If $B \neq 0$, the probability that the Monte Carlo average has the wrong sign is at most

$$\min\left\{1, 2\exp\left(-\frac{2MB^2}{W^2}\right)\right\}.$$

Proof. Condition on the observable decision state so the working distribution and containing interval are fixed. Hoeffding's inequality for independent bounded variables gives the first display. A sign disagreement requires absolute mean error at least $|B|$; substituting $\epsilon = |B|$ gives the second display. □

In the benchmark, the current term varies over an interval of width at most $\sigma = 10$. Each future use has saving at most $P - a = 10 - 1 = 9$, and at most $T - t$ future uses remain. Therefore one valid score-range width is $W = 10 + 9(T - t)$. This is a numerical-integration guarantee under DSA's working posterior, not a guarantee that the posterior model itself is correctly specified.

B Experimental Specification and Extended Results

This appendix specifies the executable benchmark and DSA, then reports candidate-quality diagnostics, strong fixed baselines, component ablations, build-cost sensitivity, and the demand-regime study. Unless otherwise noted, costs are normalized modeled units and the stream seed is the replication unit.

B.1 Benchmark and Controller Specification

API-migration benchmark. The benchmark contains 12 exact API families: four function renames, four keyword renames, and four constructor migrations. Each family has three observable coding patterns—direct use, symbol aliasing, and module aliasing—and a bank of executable Python clients against a `toyapi` package exposing the updated interface. Reusable candidates are LibCST transformations. Application and validation execute transformed clients in fresh temporary repositories; failed temporary work is discarded. Patch is represented by a reliable gold one-off repair, so policy comparisons hold final task correctness fixed.

For every family–starting-pattern pair, three one-shot candidate transformations were generated, giving 108 candidates. Generation used Codex CLI 0.153.3 with model `gpt-5.6-terra`, medium reasoning effort, and the same frozen one-shot protocol for all attempts. The candidate pool was frozen and hash-verified before policy evaluation; no candidate is regenerated or edited during any reported policy run.

Admission, scope, and service. A candidate is evaluated on the current client, two additional positive cases selected by the frozen validation rule, and all required negative regression cases. Admission requires all checks to pass. Negative cases contain unrelated same-named code that must remain unchanged. Each frozen candidate manifest declares only its starting pattern; admission does not expand that declaration. Operational reuse is therefore permitted only on that declared pattern, while hidden executable coverage on other patterns is an audit-only diagnostic.

Among multiple applicable admitted tools, the controller uses the service option with lowest posterior-mean expected cost. Tool/pattern reuse reliability uses a $\mathrm{Beta}(1, 1)$ belief updated only by executed reuse on that pattern. A detected reuse failure falls back to patch, and a tool is quarantined after three observed online reuse failures. No reuse failure occurred in the final fixed-count or demand-map studies, so this reliability-learning mechanism is implemented but not materially stressed by those streams.

DSA demand and acquisition beliefs. For an acquisition opportunity in current family f, a service cell is (f, k) for $k \in \{\text{direct}, \text{symbol-alias}, \text{module-alias}\}$. Family demand uses a Dirichlet prior with unit mass on each of the 12 families; each family has a separate three-pattern Dirichlet prior with unit mass per pattern. The current arrival is incorporated before valuation. With $R = T - t$ future periods, each numerical world samples family probabilities θ, then a remaining current-family count $n_f \sim \mathrm{Binomial}(R, \theta_f)$, within-family pattern probabilities ϕ_f, and finally $(N_k)_k \sim \mathrm{Multinomial}(n_f, \phi_f)$.

Candidate current success and admission use a joint three-category distribution over $(G, I) \in \{(0, 0), (1, 0), (1, 1)\}$. Its frozen hierarchical initialization excludes the target exact family's fold, has total concentration 3, and uses shrinkage parameter $\tau = 4$. Observed builds update this joint

Table 2: **DSA working state and observable updates.**

Object	Operational meaning	Working distribution	Observable update
Family demand	Future recurrence of exact API families	Dirichlet($\mathbf{1}$ + family counts)	Increment current family before valuation
Pattern demand	Pattern mix within family f	Dirichlet($\mathbf{1}$ + pattern counts$_f$)	Increment current pattern before valuation
(G, I)	Current success / durable admission	Three-category hierarchical Dirichlet	One observed category per actual build
U_k	Candidate transfer reliability on pattern k	$\mathrm{Beta}(1, 1)$ belief by context and pattern	Executed reuse evidence only
Tool reliability	Reliability of an admitted tool on a declared pattern	$\mathrm{Beta}(1, 1)$ initially	Executed reuse outcome
Declared scope	Patterns on which reuse is allowed	Frozen candidate manifest	Fixed after admission

state. New-candidate transfer reliability for pattern k is represented by a $\mathrm{Beta}(1, 1)$ belief U_k indexed by generation context and pattern and updated only through executed reuse evidence. Because operational reuse is restricted to declared scope, beliefs for non-declared patterns receive no online reuse updates. Under patch cost P and application cost a, the sampled transfer saving is

$$S_{A,k} = [PU_k - a]_+.$$

DSA uses $M = 512$ coupled posterior worlds and builds iff the mean of

$$G\sigma + I \sum_k N_k S_{A,k} - b$$

is strictly positive. The working score uses latent transfer value across all three pattern cells even though actual reuse remains restricted to the frozen manifest's declared starting pattern; it is therefore a predictive acquisition score rather than an exact operational marginal-value identity. The hard acquisition gate is open exactly when no active same-family tool declares the current pattern, and it prevents replacement acquisition on an already declared pattern even if patching would currently be cheaper than reuse.

Cost accounting and exhaustion. The primary modeled costs are patch $P = 10$, application $a = 1$, current successful-build saving $\sigma = 10$, and build plus validation cost $b = 19.5$. The latter corresponds to one current test, two additional positive tests, and two required negative tests under the fixed accounting. For all reported runs the disjoint cost identity is

$$C = bN_{\mathrm{build}} + aN_{\mathrm{reuse}} + PN_{\mathrm{patch}} + PN_{\mathrm{fallback}} + PN_{\mathrm{exhaust}}.$$

Here N_{build} counts actual candidate attempts only, N_{fallback} excludes exhaustion, and an empty candidate queue incurs patch cost but no build cost or fake candidate update. The patch charge for exhaustion was already included in the original cost totals; reporting exhaustion separately from other fallback therefore changes only the reporting categories, not total modeled cost. Within each family–pattern cell, the three frozen candidates are consumed according to a pre-frozen seed/context queue shared by all policies within a paired stream. Once that queue is empty, later eligible acquisition requests are served by patch, incur no build cost, and create no candidate-quality update. Thus paired policies face the same candidate population and candidate ordering, conditional on the opportunities each policy chooses to acquire.

Controlled fixed-count profiles and inference. The balanced profile has horizon $T = 72$ and exactly six arrivals per API family, followed by a seeded shuffle. The heavy-tailed profile has horizon $T = 76$ and family-count multiset

$$[20, 16, 12, 8, 6, 4, 3, 2, 2, 1, 1, 1],$$

with counts assigned to the 12 families by a seed-dependent permutation before clients are shuffled. Within each family, both profiles use the same deterministic nine-client bank ordered as direct variants v1–v3, symbol-alias variants v1–v3, and module-alias variants v1–v3. A family assigned c_f arrivals contributes the first c_f entries of a cyclic repetition of this bank before the final global shuffle; there is no additional pattern or client sampling. Consequently, the balanced profile contains three direct and three symbol-alias arrivals per family and no module-alias arrivals, while the heavy-tailed profile uses the same rule after the seed-dependent assignment of family counts. Concrete

clients repeat only when a family's count exceeds nine. These profiles control realized recurrence counts and are not i.i.d. categorical samples from the separable theory. The same stream is supplied to every policy in a paired seed. The fixed-count evaluation uses 200 previously unused seeds, 506–705. Reported paired Student-t intervals quantify stream-to-stream variability conditional on the single frozen 108-candidate population; they do not include candidate-population uncertainty.

The fixed-count profiles intentionally do not match DSA's family-level demand model. Conditional on a fixed multiset, observing another arrival of a family reduces that family's remaining count, whereas DSA continues to update a Dirichlet belief for an unknown categorical arrival distribution. The fixed-count experiments therefore test the controller under both controlled recurrence and demand-model misspecification. The separate Zipf study removes this family-level mismatch by sampling families i.i.d. from fixed Zipf probabilities, although its within-family clients follow a frozen cyclic order rather than independent pattern draws. The fact that removing demand adaptation helps in the fixed-count balanced profile is consistent with this mismatch, but the experiment does not identify it as the sole cause of that reversal.

B.2 Candidate-Pool Audit and Quality Prediction

Table 3 summarizes the complete executable audit. Current-client success is substantially more common than durable admission: 99 candidates solve the triggering client, while 77 pass the fixed admission rule. This is the empirical motivation for keeping G and I distinct.

Table 3: **Complete candidate-pool audit.** Rates use all 108 one-shot generation attempts.

Outcome	Count	Rate
Current-client success	99	91.7%
Negative-safe	102	94.4%
Admitted	77	71.3%
Same-pattern robust	99	91.7%
Any safe cross-pattern transfer	83	76.9%
Multi-pattern safe	82	75.9%
Useful beyond current client	100	92.6%

Here *negative-safe* means passing all required negative regression cases without changing the protected unrelated code. *Multi-pattern safe* means satisfying hidden positive cases across all three syntax patterns while remaining negative-safe. Admission is heterogeneous across observable contexts: 63.9% for constructor migrations, 58.3% for function renames, and 91.7% for keyword renames; by starting pattern it is 77.8% for direct use, 52.8% for module aliases, and 83.3% for symbol aliases.

The cross-fitted hierarchical prior is evaluated by holding out entire exact API families. Table 4 reports family-macro predictive metrics; lower is better. Contextual hierarchy improves admission and multi-pattern-safety prediction relative to a global prior. These predictive results motivate the initialization used by DSA; policy-level effects are evaluated separately.

Table 4: **Held-out generation-quality prediction.** Each fold holds out entire exact API families.

Target	Prior/model	Log loss	Brier
Admission	Global	0.609	0.208
Admission	Archetype	0.573	0.196
Admission	Archetype + pattern	0.531	0.181
Admission	Hierarchical ($\tau = 4$)	**0.524**	**0.180**
Multi-pattern safe	Global	0.556	0.184
Multi-pattern safe	Hierarchical ($\tau = 4$)	**0.449**	**0.143**

B.3 Fixed Baselines

All policies use the same frozen pool, candidate ordering, validation, scope semantics, service costs, and fallback rules.

Always Patch. Never acquires a candidate.

Family-First. Makes one designated acquisition attempt at the first occurrence of each exact API family and never retries that family after non-admission. After the first admission, it makes no additional family acquisition.

Family-First + Retry. Starts at the first eligible family occurrence and retries on later eligible occurrences until the family obtains its first admitted tool; after that admission it makes no additional family acquisition.

Pattern-First. For each exact family–pattern pair, makes one acquisition attempt at the first occurrence for which the standard acquisition gate is open; it never retries that pair after non-admission.

Always-Build-When-Eligible. Builds at every arrival for which DSA's standard acquisition gate is open, without a value threshold. Under this gate, augmenting Pattern-First with unconditional retry after every non-admission yields the same operational policy as Always-Build-When-Eligible: after the first gate-open occurrence of a family–pattern pair, every later gate-open occurrence triggers another acquisition until admission closes the cell or the frozen candidate queue is exhausted. The explicit unconditional-retry diagnostic reproduced Always-Build-When-Eligible's modeled costs and executed build/reuse behavior on all 200 paired streams in both fixed-count profiles; bookkeeping differs only after observed queue exhaustion.

Family-Build-After-3. Makes one family-level attempt at the designated third family occurrence and does not retry.

DSA. Uses the posterior score and strict-positive rule in Algorithm 1.

Pattern-First is the direct operational analogue of type-level Build First in the cell-separable specialization described in Section 5.4. Family-First is a coarser family-level analogue.

B.4 Fixed-Count Policy Comparisons

Table 5 gives the complete 200-stream comparison. Under heavy-tailed recurrence, DSA has mean modeled cost 583.90 versus 625.79 for Pattern-First, with paired difference -41.88 and 95% CI $[-47.05, -36.71]$; DSA is cheaper on 178 of 200 streams. Under fixed-count balanced demand, Pattern-First is cheaper: DSA costs 640.03 versus 612.46, with paired difference $+27.57$ and 95% CI $[24.15, 30.98]$. DSA's aggregate successful-reuse/admission ratio is 2.91 under heavy-tail versus 2.46 for Pattern-First; under balanced demand the corresponding ratios are 1.90 and 2.00.

Table 5: **Fixed-count balanced and heavy-tailed results.** All action counts are per-stream means over 200 paired streams. Fallback excludes exhaustion; exhausted acquisition decisions incur patch cost and are reported separately.

Demand	Policy	Cost	Builds	Admit	Reuse	Patch	Fallback	Exhaust.	Reuse/Admit
Balanced	DSA	640.03	20.555	17.200	32.655	18.790	1.865	0.000	1.899
Balanced	Always Patch	720.00	0	0	0	72.000	0	0	—
Balanced	Family-First	668.75	12.000	9.825	19.650	40.350	1.160	0	2.000
Balanced	Family-First + Retry	665.36	14.705	12.000	23.110	34.185	1.365	0	1.926
Balanced	Pattern-First	**612.46**	24.000	19.905	39.810	8.190	2.275	0	2.000
Balanced	Always-Build-When-Eligible	638.89	29.105	23.430	42.895	0	2.845	0	1.831
Balanced	Family-Build-After-3	737.79	12.000	9.950	11.945	48.055	1.130	0	1.200
Heavy-tail	DSA	**583.90**	18.955	14.270	41.530	15.365	1.760	0.150	2.910
Heavy-tail	Always Patch	760.00	0	0	0	76.000	0	0	—
Heavy-tail	Family-First	730.33	12.000	9.175	17.580	46.420	1.455	0	1.916
Heavy-tail	Family-First + Retry	719.65	14.685	11.005	21.945	39.355	1.765	0.015	1.994
Heavy-tail	Pattern-First	625.79	22.000	16.530	40.635	13.365	2.250	0	2.458
Heavy-tail	Always-Build-When-Eligible	630.24	27.995	19.995	47.085	0	2.805	0.920	2.355
Heavy-tail	Family-Build-After-3	731.25	7.000	5.395	11.400	57.600	0.735	0	2.113

The fixed-count profiles deliberately control realized family counts. Balanced fixes six occurrences per family, whereas the heavy-tailed profile concentrates many arrivals on a subset of families. This distinction is separate from the probabilistic Zipf experiment in Section B.8, where realized counts remain random.

B.5 Component Ablations

Two diagnostic removals isolate operational components of final DSA while keeping its remaining priors, gate, pool, and cost accounting fixed. *No Demand Adaptation* freezes the family and pattern demand beliefs at their initial symmetric priors for acquisition valuation. *No Retry* prevents a second acquisition on a family–pattern pair after an actual non-admitted build, while still allowing DSA to delay its first build and to acquire on other patterns.

Table 6: **Final-controller ablations.** Entries are ablation minus DSA modeled cost over the same 200 paired streams; positive values mean that removing the component increases cost.

Component removed	Balanced	Heavy-tail
Demand adaptation	$-8.72\ [-11.28, -6.15]$	$+24.80\ [19.23, 30.37]$
Retry after non-admission	$-6.28\ [-7.89, -4.67]$	$+14.16\ [9.86, 18.46]$

Both removals increase cost under heavy-tailed recurrence and decrease cost under the fixed-count balanced profile. The effects are therefore regime dependent. These are trajectory-level component interventions on DSA, not the two exact cost terms of Theorem 2, and their effects need not be additive.

A separate initialization diagnostic replaces the cross-fitted hierarchical distribution over (G,I) with a symmetric Dirichlet(1,1,1), while leaving online quality updates and all other controller components unchanged. Mean modeled cost increases from 640.03 to 697.29 under balanced demand and from 583.90 to 627.07 under heavy-tailed demand. In the heavy-tailed profile, the uninformative-prior variant differs from Pattern-First by +1.29 units, with paired 95% CI [-5.18, 7.75], so the resolved advantage of final DSA is no longer present. This intervention isolates the initialization rather than removing online quality learning, and indicates that context-dependent prior information about candidate quality materially contributes to the primary heavy-tail result.

B.6 Acquisition Composition and Subsequent Reuse

Every DSA build in the fixed-count evaluation is assigned to one of three observed categories. A *first-family* build is the first DSA acquisition attempt for that family. A *new-pattern* build is the first attempt for a family–pattern pair after earlier acquisition activity in the same family. A *retry* follows a previous non-admission on the same pair. A fourth reacquisition category was defined but observed zero.

Under heavy-tailed demand, 44.08% of DSA acquisitions are first-family attempts, 42.39% are new-pattern attempts, and 13.53% are retries. The corresponding balanced shares are 53.51%, 39.36%, and 7.13%. These categories also generate later successful reuse: under heavy-tail demand, first-family, new-pattern, and retry acquisitions are followed by 15.985, 20.990, and 4.555 successful reuses per stream, respectively. Figure 4 visualizes these quantities.

Family-level retry alone lowers heavy-tail Family-First cost by 10.68 units. Moving from one family-level attempt to one designated gate-eligible attempt per family–pattern pair lowers that baseline cost by about 104.5 units. The comparison is descriptive because the changed baseline also changes candidate contexts, timing, admission feedback, and later service.

B.7 Sensitivity to Normalized Acquisition Cost

A secondary robustness study varies only the normalized build cost, with $b \in \{0.5, 1, 2, 4\} \times 19.5$; patch cost, application cost, validation, candidate outcomes, priors, and the 200 paired fixed-count streams are unchanged. Figure 5 and Table 7 report DSA relative to Pattern-First.

Table 7: **Build-cost sensitivity.** Differences are DSA minus Pattern-First.

Multiplier	b	Balanced difference	Heavy-tail difference	DSA builds (Bal./HT)
$0.5\times$	9.75	$-21.09\ [-23.32, -18.85]$	$-53.18\ [-58.65, -47.71]$	28.665 / 26.995
$1\times$	19.50	$+27.57\ [24.15, 30.98]$	$-41.88\ [-47.05, -36.71]$	20.555 / 18.955
$2\times$	39.00	$-298.16\ [-303.85, -292.47]$	$-280.85\ [-288.40, -273.30]$	4.305 / 6.395
$4\times$	78.00	$-1296.46\ [-1301.55, -1291.37]$	$-1149.13\ [-1157.39, -1140.86]$	0.000 / 0.110

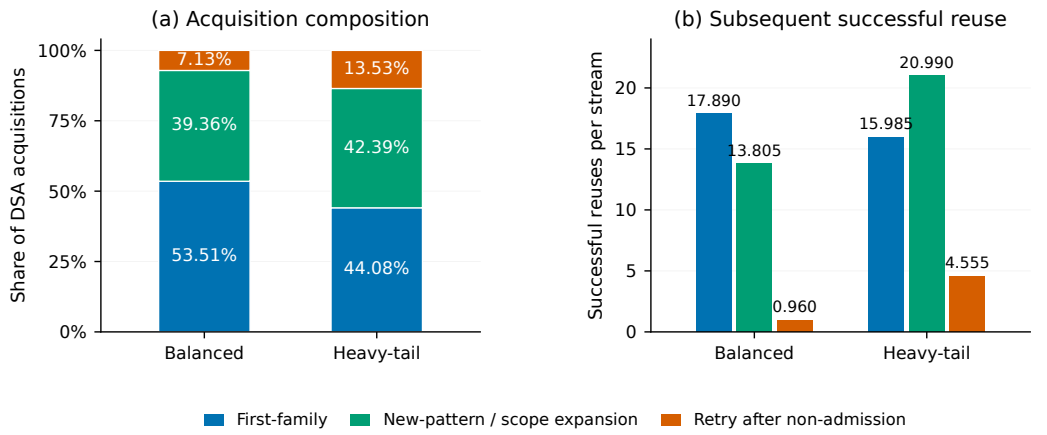


Figure 4: **DSA acquisition activity and subsequent reuse.** (a) Fractions of actual DSA acquisitions in the 200-stream fixed-count profiles. (b) Subsequent successful reuses per stream. New-pattern denotes the first attempted acquisition for a family–pattern pair after earlier acquisition activity in the family. These categories describe observed activity rather than a causal decomposition of cost savings.

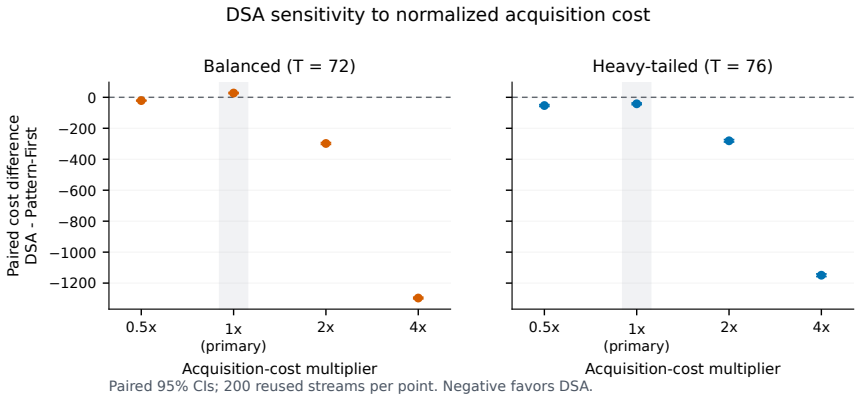


Figure 5: **Sensitivity to normalized acquisition cost.** Paired modeled-cost difference DSA minus Pattern-First as the build-cost multiplier varies; negative values favor DSA. Intervals are paired 95% Student-t intervals over the same 200 streams per point. The primary setting is $1\times$.

DSA is cheaper than Pattern-First in all four heavy-tail settings; the balanced reversal occurs only at the primary $1\times$ cost. At $0.5\times$, Always-Build-When-Eligible has the lowest mean among the five acquisition policies tested in this sensitivity study. At high build cost the large gap to Pattern-First mainly reflects avoidance of expensive first-seen acquisition: DSA exceeds the arithmetic Always-Patch cost at $2\times$ in both profiles, ties it at $4\times$ balanced, and is slightly more expensive at $4\times$ heavy-tail. Thus the sensitivity study supports the acquisition-cost dependence of the decision rather than universal dominance of DSA.

B.8 Demand Concentration and Horizon

A separate study evaluates final DSA across $\alpha \in \{0, 0.1, \ldots, 2.0\}$ and horizons $T \in \{60, 72, 120\}$. Each cell uses 100 previously unused paired stream seeds, 706–805. Exact API families are ranked by a seed-specific frozen permutation and sampled i.i.d. from a Zipf distribution with exponent α; within a family, positive clients cycle through three direct, three symbol-alias, and three module-alias

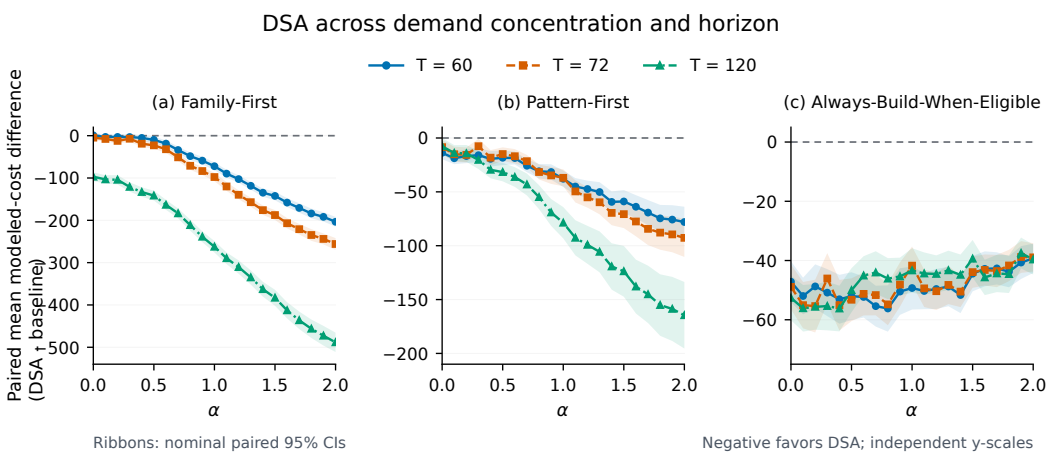


Figure 6: **DSA across demand concentration and horizon.** Paired mean modeled-cost differences relative to (a) Family-First, (b) Pattern-First, and (c) Always-Build-When-Eligible. Negative values favor DSA. Ribbons are nominal, unadjusted cellwise paired 95% intervals and panels use independent y-scales. For DSA versus Pattern-First, 61 of 63 cells remain resolved after Holm correction across the grid.

cases. This generator is distinct from the fixed-count balanced profile: at $\alpha = 0$, family probabilities are uniform but realized family counts remain random.

Against Pattern-First, DSA has lower mean cost in all 63 tested cells and all 63 nominal paired intervals exclude zero. After Holm correction across the 63 cells, 61 remain resolved; $(\alpha = 0, T = 120)$ and $(\alpha = 0.3, T = 72)$ become unresolved, and no cell is resolved in favor of Pattern-First. Against Family-First, DSA is resolved at $\alpha = 0.5$–2.0 for $T = 60$, at $\alpha = 0.1$–2.0 for $T = 72$, and throughout the grid for $T = 120$ under nominal cellwise inference. The magnitude of the differences is not universally monotone in α or T.

B.9 Scope, Coverage, and Numerical Diagnostics

Declared operational scope is intentionally distinct from hidden executable coverage. In the fixed-count evaluation, DSA's mean declared coverage is 47.78% balanced and 39.64% heavy-tail, while hidden executable coverage is 85.99% and 63.71%, respectively. Pattern-First reaches 55.29% and 45.92% declared coverage. Hidden coverage is attached only after trajectories finish and never grants operational reuse permission. This gap also makes the standalone cross-pattern transfer term in DSA a working prediction rather than a guarantee of selectable future service.

We also evaluate a scope-consistent valuation diagnostic that leaves the acquisition gate, demand beliefs, candidate-quality beliefs, retry behavior, pool, and service rules unchanged, but credits future value only to the candidate's declared starting pattern. At the primary build cost, this change reduces mean builds from 20.555 to 9.800 in the balanced profile and from 18.955 to 10.290 in the heavy-tailed profile. Mean modeled cost increases by 43.48 units (95% CI [39.00, 47.96]) under balanced demand and by 45.91 units (95% CI [39.79, 52.04]) under heavy-tailed demand.

The comparison reverses when acquisition is more expensive. At twice the primary build cost, the scope-consistent variant makes only 0.075 and 0.285 builds per stream in the balanced and heavy-tailed profiles, respectively, and lowers cost relative to final DSA by 60.62 and 13.18 units. Its absolute costs, 721.68 and 760.76, are nevertheless close to the corresponding Always-Patch costs of 720 and 760. These results characterize the broader cross-pattern term as a more aggressive predictor of latent future usefulness: it supports substantially more acquisition at the primary cost, while becoming less attractive as acquisition cost rises.

A separate numerical-integration audit compares the fixed $M = 512$ posterior calculation with an 8192-draw numerical reference on matched audit states. The 512-draw sign disagrees on 438/19,660 states (2.23%); an adaptive diagnostic reduces this to 16/19,660 (0.081%). These states come from

a separate overlap-aware precision diagnostic rather than final DSA trajectories, and the 8192-draw estimate is a high-sample numerical reference rather than an exact oracle. This diagnostic does not alter the final frozen controller, and improved numerical sign stability is not interpreted as a realized-cost guarantee.

B.10 Generation Compute and Integrity

Generating the 108-candidate pool used 13,485,824 input tokens, of which 11,679,488 were cached, and 462,744 output tokens, including 212,401 reasoning-output tokens. Scientific generation wall time was 10,313.952 seconds (2 h 51 m 53.952 s). These are provenance and compute measurements; the primary policy objective uses normalized modeled costs and does not convert the generation logs into b. Controller-side posterior-computation time was not separately measured.

The robustness studies comprise 8,800 trajectories for build-cost sensitivity and the two DSA ablations, together with 1,600 targeted diagnostic trajectories on the same 200 fixed-count seeds for scope-consistent valuation at the primary and doubled build costs, unconditional retry, and uninformative (G, I) initialization. All use the frozen candidate pool and the pre-specified controller/configuration associated with each variant; no candidate was regenerated or edited and no parameter was tuned on these runs. All tasks succeed after permitted fallback, and policy evaluation is replayable from the frozen candidate population; regenerating the original LLM-produced candidate population may additionally depend on model/service availability. The full internal repository test suite (109 tests) passed before and after the targeted diagnostics, and independent aggregation of the raw checkpoints reproduced the reported diagnostic means and paired contrasts to numerical precision. The supplementary artifact includes 27 focused tests for replay verification.

C Extended Discussion

This section places the acquisition problem in a broader literature and outlines extensions beyond the controlled API-migration setting.

C.1 Extended Related Work

Tool and skill creation. Several systems study how reusable tools or skills can be created, abstracted, and retrieved. Qian et al. (2023) organize tool creation around creation, execution, and rectification, while Yuan et al. (2024) build specialized toolsets through abstraction, validation, and retrieval. Zheng et al. (2025) synthesize and refine reusable web-agent skills, and Yue et al. (2025) study scalable tool creation and aggregation. Shen et al. (2026) reuse prior experience to move from tool use toward tool creation in multimodal reasoning. These systems broaden the evidence that reusable artifacts can improve later problem solving; our focus is the complementary pre-acquisition decision of whether the expected service value justifies paying to create and validate such an artifact.

Adaptive skill and memory management. Recent work increasingly treats persistent libraries themselves as adaptive objects. Xia et al. (2026) evolve a persistent skill bank using performance-triggered recursive updates and reinforcement learning, while the boundary-guided *SkillSmith* system of Xu et al. (2026a) compiles reusable agent skills into guarded runtime interfaces. Recurrence-sensitive memory consolidation (Dai et al., 2026), learned skill-bank management (Li et al., 2026), and the co-evolving *SkillSmith* system of Wei et al. (2026) make creation, consolidation, retention, or retirement decisions explicit. DSA addresses a narrower investment problem within this broader space: before a candidate's outcome is known, it values current service, possible durable admission, and remaining reuse demand against acquisition cost.

Broader self-evolution and allocation of search effort. A complementary family of systems improves an agent or its search process rather than acquiring a task-serving asset. AlphaEvolve uses evolutionary search to improve programs (Novikov et al., 2025); AdaEvolve adaptively allocates optimization effort (Cemri et al., 2026); and EvoX meta-evolves search strategies together with candidate solutions (Liu et al., 2026). Their resource-allocation question concerns iterative optimization and search progress, whereas the persistent-tool setting asks whether an acquisition expenditure can be amortized by later service of recurring tasks.

Program transformations, repair, and reusable code-update rules. The API-migration setting has substantial pre-LLM and LLM-based transformation literature. Ramos et al. (2023) mine reusable lightweight transformations from pull requests, and Bader et al. (2019) learn reusable bug-fix patterns with contextual ranking. Dilhara et al. (2024) combine LLMs with transformation by example, while Cummins et al. (2024) generate transformation procedures instead of applying isolated edits. The BUMP benchmark provides reproducible breaking dependency updates (Reyes et al., 2024), and Reyes et al. (2026b) use LLMs to repair such updates directly. Ramos et al. (2026) synthesize SPELL transformations intended for reuse, and the AST-transformation work of Reyes et al. (2026a) likewise produces reusable executable transformations. These papers motivate executable migration tools and validation-based reuse; the sequential question here is whether to incur acquisition cost before reusable quality and future recurrence are known.

Migration benchmarks. Existing benchmarks provide complementary collections of real-world migration tasks. PyMigBench contains Python library-migration changes, while CODEMENV evaluates code migration across package and environment changes (Islam et al., 2023; Cheng et al., 2025). These benchmarks emphasize realistic migration instances and model capability, whereas our controlled benchmark is designed around sequential recurrence, reusable candidate acquisition, fixed validation and admission, and matched task streams across policies. Adapting such datasets to the present setting would additionally require defining recurring capability types, reusable transformations, validation rules, and operational scope, making them natural targets for broader empirical transfer rather than drop-in replacements for the controlled study.

Amortization and break-even analysis. A smaller literature studies whether expensive model-based optimization or compilation can repay its upfront cost through repeated execution. Peng et al. (2026) discuss that the upfront cost of LLM-based system optimization must be weighed against savings over later executions, while Coignion et al. (2025) explicitly study a break-even point for the energy cost of LLM-based code optimization. A controlled real-cost study of language-model agent skill optimization similarly separates one-time compilation cost from downstream execution (Xu & Wu, 2026). These works provide useful precedents for amortization; DSA makes the purchase decision online under uncertain recurrence and stochastic admission.

Online investment and distribution-sensitive rent-or-buy. Beyond multislope ski rental (Lotker et al., 2008), related online-investment models include the parking permit problem, where purchases cover future demand for limited durations (Coester & Turoczy, 2026), and multi-commodity ski rental, where several rent/buy decisions interact (Wu et al., 2022). Distributional information has also been incorporated into online algorithms: Madry & Panigrahi (2011) study semi-stochastic ski rental with partial distributional access, Diakonikolas et al. (2021) learn online algorithms from distributional advice, and Cui & Dinitz (2026) study ski rental with distributional predictions of unknown quality. These models clarify durable acquisition and uncertain demand. The additional uncertainty here is whether an acquisition produces a capability admitted for future service at all.

Persistent action generation and discovery. In online learning, dynamic arm acquisition (Kalvit & Zeevi, 2022) and ballooning bandits (Ghalme et al., 2020) study enlarging action sets, while bandits with knapsacks study learning under consumable resources (Badanidiyuru et al., 2013). The closest persistent-action precursor is Xu et al. (2026b), which models costly generation and reuse of persistent actions, chooses generation through an optimistic online-learning rule, and derives sublinear-regret guarantees. A newly generated action in that model is matched to its triggering context. The present paper instead keeps a reliable patch alternative, separates stochastic current success from durable admission, permits retries after non-admission, and isolates recurring capability types for an exact finite-horizon decision characterization. The executable evaluation further realizes acquisition outcomes as generated and validated program transformations. Concurrent cost-aware tool-selection work such as CAM-DF studies how many already-ranked tools to expose under heterogeneous costs (Feng et al., 2026). That stopping problem selects from an existing tool set, whereas our acquisition decision pays to create and validate a persistent capability whose reusable quality and future recurrence are initially uncertain.

Overlapping service and irreversible investment. The general library structure is related to persistent covering and facility-acquisition problems. Online facility location opens durable resources

as demand arrives (Meyerson, 2001), and online set cover acquires overlapping sets to cover arriving elements (Alon et al., 2003). Real-options models formalize the value of waiting before irreversible investment (McDonald & Siegel, 1986; Pindyck, 1993). The information distinction is especially relevant here: task arrivals may reveal recurrence without creating a tool, whereas evidence about candidate quality may require paid generation and validation. Proposition 5 gives a discrete example in which such free demand information creates option value beyond posterior-mean one-shot acquisition scoring.

C.2 Limitations and Extensions

Beyond API migration. API migration provides a controlled setting in which reusable capabilities are executable, testable, and naturally recurrent, but the same acquisition problem arises in many other agent workflows. An important next step is to study persistent skills for broader coding, data-analysis, web, and decision-making tasks, where task similarity and transfer scope are less explicit and must themselves be inferred.

Inferred task similarity and scope. The benchmark exposes API family and coding pattern to the controller and uses declared scope to determine whether an admitted tool may be invoked. In broader agent environments, these objects may need to be inferred from natural-language tasks, code, interaction histories, or embeddings. A natural extension is to learn task similarity and operational scope jointly with acquisition value, allowing the controller to trade off uncertain transfer against the cost of testing a capability in a new region. The benchmark also assumes a known finite family set and negligible retrieval or matching overhead. In more open-ended deployments, recurring task categories may themselves need to be discovered online, and locating applicable tools may carry nontrivial computational or latency cost; both effects could be incorporated into a richer acquisition state and service model.

Capability lifecycles. The main model focuses on acquisition and reuse after admission. More autonomous libraries may refine a tool after failures, extend its scope, compose several tools, replace an obsolete capability, or retire assets whose maintenance cost exceeds their value. These operations create a richer lifecycle problem in which the decision is not only whether to acquire a capability, but also when to upgrade, merge, or discard it.

Changing demand. The exact theorem uses stationary recurring task types to isolate acquisition timing. Long-running deployments can instead exhibit seasonality, drift, new task classes, or abrupt changes after software and environment updates. In such settings, recurrence estimation should discount stale evidence or model changes explicitly, and acquisition decisions should account for possible decay in future demand.

Online generation and feedback. The executable study uses a frozen candidate population so that policies face the same reproducible candidate outcomes. A live agent could instead change prompts, generators, validation budgets, or refinement strategies after observing previous attempts. Treating generator choice as an action would couple capability acquisition with active information gathering and tool improvement.

Richer resource objectives. Our modeled cost aggregates patching, application, validation, and acquisition into common units at fixed final correctness. Real deployments may care separately about latency, token usage, human review, monetary cost, energy, or risk. The same investment formulation can incorporate such quantities through calibrated costs or constrained objectives, but the appropriate conversion is deployment dependent. Acquisition cost itself may also be stochastic or only partially known before generation, for example when token usage or latency is realized only after a build begins. A richer controller could maintain a predictive distribution over build cost and incorporate validation risk or safety penalties directly into the acquisition objective or through explicit constraints.

Broader empirical transfer. The stream-holdout evaluation changes task streams while retaining the same frozen 108-candidate population. Broader evaluation could freeze DSA before generating disjoint API families, repository-scale migrations, or candidates from different foundation models,

thereby testing transfer across both capability populations and generators rather than only across recurrent task streams.