Constructive Specification for Plug-and-Play Learnware Agents

Jian-Dong Liu†
SinapisAI
Nanjing, China

Zi-Chen Zhao
Nanjing University
Nanjing, China

Hao Sun
China Mobile
Beijing, China

Lin-Xing Wu†
SinapisAI
Nanjing, China

Huan Zhang
China Mobile
Beijing, China

Pengyuan Wang
Nanjing University
Nanjing, China

Ming Zhao
China Mobile
Beijing, China

Xinyu Chu
Nanjing University
Nanjing, China

Shu Yan
Nanjing University
Nanjing, China

Yongbei Zhu
China Mobile
Beijing, China

Weijun Zhong
China Mobile
Beijing, China

Zhi-Hao Tan
Nanjing University
Nanjing, China

Jing Shang
China Mobile
Beijing, China

Yang Yu
Nanjing University
Nanjing, China

Zhi-Hua Zhou*
Nanjing University
Nanjing, China

Abstract

Large language models are increasingly deployed at scale as API-accessible, tool-augmented agents, forming a heterogeneous, fast-evolving agent ecosystem. A central challenge is *query-level identification*: selecting the most suitable agent per query from candidates provided as black-box services, where costly input-output evaluation makes exhaustive profiling and router retraining impractical at scale. Predating LLMs, the *learnware paradigm* provides a principled perspective on this challenge by advocating capability *specifications* and reducing identification to specification matching, avoiding pool-dependent retraining and exhaustive supervision. We operationalize it with *constructive specification*, which builds hierarchical capability representations from limited profiling over diverse benchmarks, using an optimism-guided profiler that prioritizes informative regions and prunes low-utility areas with guarantees. At serving time, we enrich query context with system-maintained benchmarks and map queries into the same specification space for multi-granularity similarity matching, enabling plug-and-play identification without accessing agent internals or training any additional selector. Experiments show that our approach, selecting among lightweight agents, outperforms contenders and matches or surpasses much larger models on several tasks.

CCS Concepts

• **Computing methodologies → Machine learning**; • **Information systems → Information systems applications**.

Keywords

Machine Learning, Learnware, Agent, Constructive Specification, Learnware Dock System, Plug-and-Play Agents

*Corresponding author (email: zhouzh@lamda.nju.edu.cn).
†Also affiliated with Nanjing University.
Institution details: Nanjing University (National Key Laboratory for Novel Software Technology and School of Artificial Intelligence), and China Mobile (Jiutian Artificial Intelligence Technology (Beijing) Co., Ltd.).

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
KDD 2026, Jeju Island, Republic of Korea.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2259-2/2026/08
https://doi.org/10.1145/3770855.3817996

ACM Reference Format:
Jian-Dong Liu, Zi-Chen Zhao, Hao Sun, Lin-Xing Wu, Huan Zhang, Pengyuan Wang, Ming Zhao, Xinyu Chu, Shu Yan, Yongbei Zhu, Weijun Zhong, Zhi-Hao Tan, Jing Shang, Yang Yu, and Zhi-Hua Zhou. 2026. Constructive Specification for Plug-and-Play Learnware Agents. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD 2026), August 9–13, 2026, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3770855.3817996

1 Introduction

The recent surge of large language models (LLMs) [3, 7, 27] and their rapid deployment as API-accessible agents [16, 39, 43] has shifted how machine learning systems are built and used. Instead of training a new model for every task, developers now wrap general-purpose LLMs into specialized agents that integrate external tools, structured workflows, or specific prompting strategies. These agents are proliferating rapidly across open platforms and closed systems, creating a rich ecosystem of capabilities. As the number and diversity of agents grow, a pressing challenge emerges: given a new user query, how can we identify the most suitable one from a large and evolving agent system, quickly, accurately, and without incurring excessive computational cost?

The *learnware paradigm* [45, 46] was proposed long before the appearance of LLMs, but it can be well utilized to tackle the aforementioned challenge. "Learnware = Model + Specification", i.e., a learnware comprises a well-performing model of any structure together with a specification that characterizes its utility and properties. Developers worldwide can submit various models to a *learnware dock system* (LDS) [37], which supports specification generation on the developer side to form learnwares without disclosing raw training data. Given a new user task, the user submits her request in the form of a specification, and the LDS automatically identifies and assembles helpful learnwares based on specifications. The user can then apply these learnwares directly or refine them with her own data to address the task. Importantly, throughout the whole process, the LDS has no access to the raw data of either model developers or users.

Prior learnware work has demonstrated effectiveness on *task-level* reuse [22, 36, 38], namely identifying helpful models for an entire task or dataset, and it typically constructs specifications based on the model's training data. API-accessible agent repositories, however, differ from this setting in two key respects. First, agents

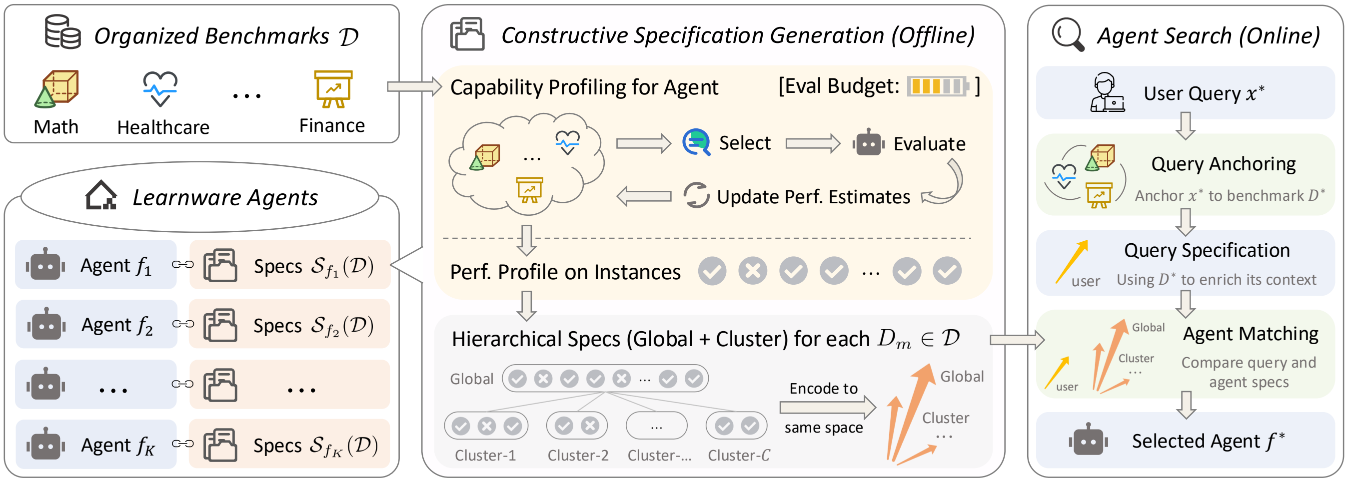

Figure 1: System overview of constructive specification for plug-and-play learnware agents.

are often exposed as black-box services, and the platform has no per-agent training data and limited supervision to construct specifications at scale. Second, the selection objective is inherently *query-level*, since the best agent may vary across queries even within the same user and task context. In continuously evolving agent systems, these differences are not directly addressed by existing learnware methods, and call for extending *learnware* to support black-box, query-level agent identification.

In addition to learnware methods, two alternative lines of work have been explored for agent or model selection, but neither fits the practical constraints of black-box agents and an evolving agent system. Manually written or language-based descriptions [30, 31] underpin semantic specifications, yet such descriptions are often too coarse and can diverge from actual capabilities. Router-based approaches train a dedicated selector [26, 47] using labeled or preference data collected over a target model pool. As the model pool scales and evolves, required supervision can increase substantially, since new candidates need to be compared against existing ones to learn relative strengths on representative queries, often followed by re-training or re-calibration. As a result, we still lack a scalable way to derive capability-based specifications from black-box agents and support plug-and-play identification without retraining.

To address these challenges, we propose *constructive specification*, which derives capability representations from demonstrated agent performance rather than textual claims or labeled preferences. An overview is shown in Figure 1. The core idea is to encode an agent's performance profile into *parameter vectors* [32, 38], which capture capability as profiling-induced parameter changes of a shared pre-trained model and support similarity comparisons in a unified specification space without accessing agent internals. We further design a *hierarchical* structure for these vectors, motivated by the observation that a single global specification cannot adequately capture heterogeneous capabilities. An agent may excel at certain problem types while struggling with others even within the same domain. We therefore decompose specifications from benchmark-level aggregates down to cluster-specific patterns, enabling more precise matching. To avoid exhaustive evaluation, we develop a budget-efficient profiling algorithm guided by the principle of *optimism in the face of uncertainty*, which dynamically explores promising regions based on optimistic estimates from evaluated neighbors. This design enables accurate capability characterization under limited budgets while remaining fully compatible with black-box agents.

For online agent selection at serving time, we design a specification-based search mechanism that handles incoming queries without any model retraining. Since user queries are typically under-specified, we enrich them with system-maintained benchmarks and then project them into the same specification space as agents for multi-granularity similarity matching. This design enables plug-and-play identification: new agents join by computing their specifications offline, and queries are matched against the entire system instantly without additional supervision or retraining.

We summarize our main contributions as follows:

- **Hierarchical constructive specification.** For large, evolving agent systems with black-box agents, we propose *constructive specification*, building hierarchical capability *representations* that capture within-agent heterogeneity in a unified specification space. Queries are projected into this space for multi-granularity matching, enabling plug-and-play identification as the system continuously evolves.
- **Budget-efficient agent profiling.** An optimism-guided active profiling algorithm is proposed to build reliable specifications under tight evaluation budgets, prioritizing promising regions via optimistic neighbor estimates and pruning low-utility areas with theoretical guarantees during iterative capability exploration.
- **Evaluation across scenarios and shifts.** Extensive experiments show that our approach, selecting among lightweight agents, outperforms contenders and matches or surpasses much larger models on several tasks. Unseen-domain results further confirm the robust out-of-distribution identification performance of our approach.

2 Problem Formulation

This section formalizes agent identification from a system perspective. Consider a learnware dock system hosting a collection of agents $\mathcal{F} = \{f_1, f_2, \ldots, f_K\}$, where each agent f maps a query $x \in \mathcal{X}$ to a response $f(x) \in \mathcal{Y}$. The goal is to select the most suitable agent from the system for each user query.

Two practical constraints shape this problem: *black-box access* and *continual evolution*. Agents are typically exposed only via input-output interfaces, with no access to parameters, architectures, or training data. This rules out approaches that require model internals (e.g., weight inspection or gradient-based analysis), leaving input-output behavior as the basis for capability characterization. Meanwhile, agents may join or leave over time, making a centrally trained selector brittle. The system therefore should be *plug-and-play*: it integrates new agents by computing their specifications offline without retraining any global component, and removing an agent does not affect selection for the remaining agents.

These constraints naturally lead to a *benchmark-driven* approach for capability characterization. The system maintains *reference benchmarks* $\mathcal{D} = \{D_1, D_2, \ldots, D_M\}$, where each benchmark $D_m = \{(x_i, y_i)\}_{i=1}^{N_m}$ consists of query-answer pairs from a specific domain (e.g., mathematics, medicine, finance). These benchmarks serve as the common ground for comparing heterogeneous agents: since agent internals are inaccessible, observed performance on shared tasks becomes the sole basis for capability comparison. Moreover, benchmarks enable plug-and-play integration, allowing new agents to be profiled offline without retraining any global component.

However, the scale of modern agent systems makes exhaustive profiling infeasible. With many agents and large benchmark collections, profiling every agent on every benchmark sample would require an impractical number of agent calls. We therefore assume a *profiling budget* B such that, for any agent f and benchmark D, at most B agent calls are allowed for profiling. The key question becomes how to allocate this limited budget to obtain effective capability representations that support accurate identification.

We formalize the *agent identification problem*. Given a user query x^*, the goal is to select an agent $f^* \in \mathcal{F}$ that maximizes response quality. Let $Q : \mathcal{Y} \times \mathcal{Y} \to [0, 1]$ denote a task-specific quality function comparing a response with the reference answer. Formally,

$$f^* = \arg\max_{f \in \mathcal{F}} \mathbb{E}\left[Q\left(f(x^*), y^*\right)\right],$$

where the expectation accounts for stochasticity in agent responses. This setting is inherently query-level and operates under black-box access and *per-benchmark* profiling budgets, motivating constructive specification $\mathcal{S}_f(\mathcal{D})$ built from budgeted observations over a system-maintained benchmark pool $\mathcal{D}$.

3 Proposed Approach

We propose *constructive specification* for query-level identification in evolving learnware agent systems. Under a strict profiling budget, we actively identify the benchmark instances each agent reliably solves from a large system-maintained pool and encode them as structured specifications in a shared vector space. This reduces agent selection to similarity matching between specifications, without retraining a centralized router each time the system evolves.

Our method consists of an offline construction stage and an online search stage. Offline, for each agent f, we construct its specification by budget-efficiently exploring the benchmark pool $\mathcal{D}$ to identify strong regions and compiling them into hierarchical specifications. Online, given a user query x^*, we retrieve relevant benchmark instances to form a query specification τ_{x^*} in the same space and rank agents by multi-granularity similarity with confidence-aware aggregation. Section 3.1 presents specification construction, and Section 3.2 describes specification-based agent search.

3.1 Constructive Specification

Constructive specifications are *built* from demonstrated agent performance on system-maintained benchmarks, rather than textual descriptions or manually defined tags. For each benchmark $D \in \mathcal{D}$, we first perform *active capability profiling* to identify samples the agent reliably solves under a strict budget, and then compile these successes into a structured representation for similarity matching.

Definition 3.1 (Constructive Specification)**.** Given a black-box agent f and a system-maintained benchmark pool $\mathcal{D} = \{D_m\}_{m=1}^{M}$, a *constructive specification* is an agent representation $\mathcal{S}_f(\mathcal{D})$ built without access to f's training data or internals, using input–output evaluations on benchmark instances and system-shared artifacts (e.g., a fixed base model or encoder), such that $\mathcal{S}_f(\mathcal{D})$ lies in a *unified, agent-independent* space for comparing heterogeneous agents.

Definition 3.2 (Benchmark-Level Specification)**.** For a benchmark $D \in \mathcal{D}$, the *benchmark-level specification* $\mathcal{T}_f(D)$ is constructed from the profiled set $\mathcal{P}_f(D)$ in Eq. (2). If $|\mathcal{P}_f(D)| < n_{\min}$, set $\mathcal{T}_f(D) = \emptyset$, where $n_{\min}$ is the minimum profile size for a reliable specification.

In this work, we instantiate $\mathcal{T}_f(D)$ as the hierarchical parameter vector set in Eq. (4), and define the agent-level specification as

$$\mathcal{S}_f(\mathcal{D}) \;=\; \{\, \mathcal{T}_f(D) \mid D \in \mathcal{D},\ \mathcal{T}_f(D) \neq \emptyset \,\}. \tag{1}$$

As agents perform differently across benchmarks, $\mathcal{S}_f(\mathcal{D})$ is variable in size: benchmarks where f has too few successful samples yield $\mathcal{T}_f(D) = \emptyset$ and are omitted. Below we describe how to construct $\mathcal{T}_f(D)$ for a benchmark D. Repeating the procedure for each $D \in \mathcal{D}$ yields $\mathcal{S}_f(\mathcal{D})$ under per-benchmark budget constraints.

3.1.1 Active Capability Profiling. The foundation of constructive specification is the *performance profile*, defined as the subset of benchmark samples that an agent handles well:

$$\mathcal{P}_f(D) = \{(x, y) \in D \mid Q(f(x), y) \geq \delta\}, \tag{2}$$

where $\delta \in (0, 1]$ is a quality threshold. Naïvely computing this profile requires evaluating agent f on all N benchmark samples, which is prohibitive for large benchmarks or costly invocations. We therefore develop an active profiling algorithm that discovers high-performing samples under a limited query budget $B \ll N$.

Benchmark Preprocessing. Before profiling, we preprocess each benchmark D to allocate budget to informative regions. Using a fixed baseline model, we split samples into *easy* (baseline succeeds) and *hard* (baseline fails). Since easy cases are typically abundant and redundant, we run K-Medoids on the embeddings of easy samples and keep one representative sample per cluster (e.g., the sample closest to each centroid), while retaining all hard samples. This preserves all baseline-failure cases and reduces repeated evaluation on near-duplicate easy instances.

Optimism-Guided Active Search. Our search strategy follows the principle of *optimism in the face of uncertainty* [2]: we prioritize candidates that could plausibly achieve high performance given current observations. Let $E(\cdot)$ denote the embedding function and $d(\cdot,\cdot)$ denote a distance induced by embeddings (e.g., Euclidean distance or some variants of cosine similarity). For convenience, let $u_f(x) := Q(f(x), y)$ denote the performance score of agent f on sample (x, y). After evaluating a set $\mathcal{V}$ of samples and obtaining $\{u_f(x)\}_{x\in\mathcal{V}}$, we assign each unevaluated candidate x' an optimistic score as a *tight* upper envelope induced by $\mathcal{V}$:

$$U_f(x') \;=\; \min\nolimits_{x\in\mathcal{V}} \Big(u_f(x) + \beta \cdot d(x', x)\Big),$$

where $\beta > 0$ controls the exploration bonus. Intuitively, larger distances contribute a larger bonus term, favoring candidates in less-covered regions (all else equal). This construction is standard in *Lipschitz/metric bandits* [4, 18]: under mild conditions, $U_f(x')$ upper-bounds $u_f(x')$ for any $x' \in D$ as shown in Theorem 3.3.

For implementation, we maintain $U^*(x)$ as an efficiently updatable upper-bound estimate that tightens as $\mathcal{V}$ grows, and prioritize candidates with the largest maintained upper bound. We start from k diverse seeds to obtain coarse coverage, and then iteratively refine the envelope by expanding kNN neighborhoods around samples exceeding the threshold δ. Candidates with upper bounds below δ are never prioritized, concentrating budget on regions that are both promising (high observed u_f) and under-explored (large distance bonus). Algorithm 1 summarizes the procedure.

Theoretical Analysis. We analyze Algorithm 1 from the perspective of *Lipschitz/metric bandits* [4, 18]: the arm space is the benchmark instances and the reward function $u_f(\cdot)$ is locally smooth in the embedding-induced metric. Our analysis establishes a logical chain of guarantees: (1) the optimistic envelope U_f upper-bounds the true performance, (2) the threshold test enables *safe pruning* without false negatives, and (3) the lazy implementation preserves these guarantees while keeping the computational overhead budget-sensitive. All proofs are deferred to Appendix B.

Theorem 3.3 (Upper-Bound Property). *Suppose u_f is L-Lipschitz w.r.t. $d(\cdot,\cdot)$: $|u_f(x) - u_f(x')| \le L \cdot d(x, x')$ for all $x, x' \in \mathcal{X}$. If $\beta \ge L$, then for any evaluated set $\mathcal{V}$ and any sample $x' \in D$,*

$$u_f(x') \le U_f(x') \;=\; \min\nolimits_{x\in\mathcal{V}} \Big(u_f(x) + \beta \cdot d(x', x)\Big).$$

Theorem 3.3 shows that $U_f(\cdot)$ is a valid optimistic envelope: the true performance never exceeds its optimistic estimate.

Corollary 3.4 (Safe Threshold Pruning). *Under the same condition as in Theorem 3.3, if an unevaluated sample x' satisfies $U^*(x') < \delta$, then it cannot meet the quality threshold, i.e., $u_f(x') < \delta$.*

Corollary 3.4 directly establishes the safety of the threshold check ($U^*(x') \ge \delta$) in Algorithm 1: any candidate whose maintained upper bound already falls below δ cannot enter the performance profile and is safely excluded from further prioritization.

Proposition 3.5 (Budget-Sensitive Queue Complexity). *Algorithm 1 dynamically maintains the upper bounds using a lazy priority queue: it may push multiple keys for the same candidate as $U^*(\cdot)$ tightens, and discards stale entries by checking $U_{key} = U^*(x)$ upon popping. Across the entire run, the number of successful tightenings (updates with $U_{new} < U^*(x')$) is at most $n|\mathcal{V}|$. Consequently,*

$$\#\text{Push} \le k + nB, \qquad \#\text{PopMax} \le k + nB,$$

and the total heap time complexity is $O\big((k + nB)\log(k + nB)\big)$ (excluding the cost of kNN search).

Algorithm 1: Budget-Efficient Capability Profiling

Input: Benchmark $D = \{(x_i, y_i)\}_{i=1}^N$ with embeddings $E(\cdot)$, agent f, budget B, threshold δ, seed clusters k, neighbors n, exploration coefficient β, distance $d(\cdot,\cdot)$.
Output: Performance Profile $\mathcal{P}_f(D)$.

Obtain D' by benchmark preprocessing in Section 3.1.1;
Initialize $Q \leftarrow \emptyset,\ \mathcal{V} \leftarrow \emptyset,\ u_f(\cdot) \leftarrow \emptyset$;
Initialize $U^*(x) \leftarrow +\infty$ implicitly for unseen $x \in D'$;
$\mathcal{S}_{\text{init}} \leftarrow$ K-Medoids$(\{E(x) \mid (x, y) \in D'\}, k)$;
Push$(Q, (U^*(x), x))$ for all $x \in \mathcal{S}_{\text{init}}$;
while $|\mathcal{V}| < B$ ***and*** $Q \neq \emptyset$ **do**
  $(U_{\text{key}}, x) \leftarrow$ PopMax(Q);
  if $x \in \mathcal{V}$ ***or*** $U_{key} \neq U^*(x)$ **then continue**;
  Evaluate f on $(x, y) \in D'$ to get $u_f(x)$, then add x to $\mathcal{V}$;
  if $u_f(x) < \delta$ **then continue**;
  foreach $x' \in$ k-NN(x, D', n) **do**
    if $x' \in \mathcal{V}$ **then continue**;
    $U_{\text{new}} \leftarrow u_f(x) + \beta\, d(x', x)$;
    if $U_{new} < U^*(x')$ **then**
      $U^*(x') \leftarrow U_{\text{new}}$;
      if $U^*(x') \ge \delta$ **then**
        Push$(Q, (U^*(x'), x'))$;
      end
    end
  end
end
return $\mathcal{P}_f(D) = \{(x, y) \in D \mid x \in \mathcal{V},\ u_f(x) \ge \delta\}$

Proposition 3.5 complements Corollary 3.4 by showing that lazy maintenance of $U^*(\cdot)$ supports safe pruning with controlled overhead: the profiling runtime scales mainly with evaluation budget B and neighborhood size n, rather than benchmark size N.

3.1.2 Hierarchical Specification. Given the actively profiled performance set $\mathcal{P}_f(D)$, we compile it into a compact specification in a unified space for reliable similarity-based matching across heterogeneous agents, even with extremely limited user-side signals.

Encoding with Parameter Vector (PAVE). We adopt PAVE to encode successfully solved benchmark instances as a single vector in a shared space [32, 38]. Given a fixed base model $h(\cdot;\theta_0)$, the PAVE specification is the *parameter update* that best fits the agent capability profile $\mathcal{P}_f(D)$ when applied to h:

$$\boldsymbol{\tau}_f(D) = \arg\min_{\boldsymbol{\tau}} \sum\nolimits_{(x,y)\in\mathcal{P}_f(D)} w(x)\, \mathcal{L}(h(x;\theta_0 + \boldsymbol{\tau}),\ y)\,, \quad (3)$$

where $w(x)$ is an optional sample weight and $\mathcal{L}$ is the training loss. For efficiency, $\boldsymbol{\tau}_f(D)$ can be approximated in a LoRA [12] manner

Algorithm 2: Hierarchical PAVE Specification Generation

Input: Profile $\mathcal{P}_f(D)$, base model $h(\cdot;\theta_0)$, baseline results, weights $(w_{\text{hard}}, w_{\text{easy}})$, constants $C_{\max}$, $n_{\min}$.
Output: Hierarchical specification set $\mathcal{T}_f(D)$ in Eq. (4).

if $|\mathcal{P}_f(D)| < n_{\min}$ **then return** $\emptyset$;
foreach $(x_i, y_i) \in \mathcal{P}_f(D)$ **do**
  if *baseline fails on* x_i **then** $w_i = w_{\text{hard}}$ **else** $w_i = w_{\text{easy}}$;
end
$\boldsymbol{\tau}_f(D) \leftarrow \text{PAVE}(\mathcal{P}_f(D), \textsc{Normalize}(\{w_i\}), \theta_0)$ by Eq. (3);
$\mathcal{T}_f(D) \leftarrow \{\boldsymbol{\tau}_f(D)\}$, $C \leftarrow \min(C_{\max}, \lfloor|\mathcal{P}_f(D)|/n_{\min}\rfloor)$;
if $C \leq 1$ **then return** $\mathcal{T}_f(D)$;
$\{c_i\} \leftarrow \textsc{K-Means}(\{E(x_i)\}, C)$;
foreach $c \in \{1, \ldots, C\}$ **do**
  $\mathcal{P}_f(D, c) \leftarrow \{(x_i, y_i) \in \mathcal{P}_f(D) \mid c_i = c\}$;
  if $|\mathcal{P}_f(D, c)| < n_{\min}$ **then continue**;
  Compute $\boldsymbol{\tau}_f(D, c)$ by solving Eq. (3) on $\mathcal{P}_f(D, c)$;
  $\mathcal{T}_f(D) \leftarrow \mathcal{T}_f(D) \cup \{\boldsymbol{\tau}_f(D, c)\}$;
end
return $\mathcal{T}_f(D)$

via low-rank updates [32]. All specifications are defined relative to the same θ_0, so $\boldsymbol{\tau}_f(D)$ vectors are directly comparable across agents, enabling efficient similarity matching (e.g., via cosine similarity).

Discriminative Sample Weighting. Not all successful samples are equally informative: ubiquitous successes contribute little, while harder ones better separate capabilities. We reuse the baseline difficulty tags (easy vs. hard) in Section 3.1.1 and set $w_i = w_{\text{hard}}$ if the baseline fails on x_i, and $w_i = w_{\text{easy}}$ otherwise, with $w_{\text{hard}} > w_{\text{easy}}$. These weights instantiate $w(x)$ in Eq. (3) for PAVE specifications. For stability, we normalize weights within each set to keep the objective scale comparable across profiles.

Hierarchical PAVE Specification. A single global PAVE $\boldsymbol{\tau}_f(D)$ can obscure within-domain heterogeneity. This is especially problematic at the query level: an agent may excel at one semantic sub-type but fail at another, while the user provides only a small amount of query-side evidence. To make matching robust under such sparse signals, we refine the flat PAVE representation into a hierarchical set that preserves fine-grained capability structure.

We partition the performance profile into C semantic clusters via K-Means on sample embeddings, and extract a parameter vector for each sufficiently large cluster:

$$\mathcal{T}_f(D) = \{\boldsymbol{\tau}_f(D)\} \cup \{\boldsymbol{\tau}_f(D, c)\}_{c \in C_f}, \tag{4}$$

where $C_f = \{c : |\mathcal{P}_f(D, c)| \geq n_{\min}\}$ indexes valid clusters and $\boldsymbol{\tau}_f(D, c)$ is computed by applying PAVE to $\mathcal{P}_f(D, c) \subseteq \mathcal{P}_f(D)$. We choose C adaptively as $C = \min(C_{\max}, \lfloor|\mathcal{P}_f(D)|/n_{\min}\rfloor)$. Algorithm 2 summarizes the complete generation process.

Repeating this process for each $D \in \mathcal{D}$ yields the agent-level specification $\mathcal{S}_f(\mathcal{D})$ defined in Eq. (1), which supports plug-and-play comparison across heterogeneous black-box agents.

3.2 Specification-Based Agent Search

Given a user query x^*, we project it into the same parameter-vector space as agent specifications and perform similarity-based selection.

Algorithm 3: Specification-Based Agent Search

Input: Query x^*, benchmarks $\mathcal{D}$, agent specs $\{\mathcal{S}_f(\mathcal{D})\}$, decay weights $\{\omega_j\}_{j=1}^{k}$, constants $\alpha_{\min}$, $\alpha_{\max}$.
Output: Ranked agents $\{(f, \text{score}_f)\}$.

Anchor the query to the most relevant benchmark D^* via Eq. (5) and get the local neighborhood $\mathcal{R}(x^*)$;
Construct the query specification $\boldsymbol{\tau}_{x^*}$ by Eq. (6);
foreach $f \in \mathcal{F}$ **do**
  if $\mathcal{T}_f(D^*) = \emptyset$ **then continue**;
  Compute s_f by Eq. (7) over $\{\cos(\boldsymbol{\tau}_{x^*}, \boldsymbol{\tau}) : \boldsymbol{\tau} \in \mathcal{T}_f(D^*)\}$;
end
Min–max normalize $\{s_f\}, \{g_f(D^*)\}$ to obtain $\{\bar{s}_f\}, \{\bar{g}_f\}$;
$\alpha \leftarrow \text{clip}((\max_f s_f + 1)/2,\ \alpha_{\min},\ \alpha_{\max})$;
Compute score_f by Eq. (8) for each agent f;
return $\{(f, \text{score}_f)\}$ *sorted by* score_f *descending*;

Since specifications are organized per benchmark as $\mathcal{T}_f(D)$, we first anchor the query to a relevant benchmark and then match within that benchmark's specification space.

Benchmark Anchoring. For each benchmark D_m, we retrieve a local neighborhood $\mathcal{R}_{D_m}(x^*) \subseteq D_m$ via vector search, consisting of its top-m nearest samples to x^* under embedding similarity. We score coverage by averaging similarities within the neighborhood:

$$\text{rel}(D_m, x^*) = \frac{1}{m}\sum\nolimits_{(x,y)\in\mathcal{R}_{D_m}(x^*)} \text{sim}\big(E(x^*), E(x)\big), \quad D^* = \arg\max\nolimits_{D_m \in \mathcal{D}}\ \text{rel}(D_m, x^*). \tag{5}$$

Averaging over multiple neighbors reduces sensitivity to noise.

Query Specification. A single query x^* is typically short and under-specified, making its direct projection into the specification space unstable and unreliable. To mitigate this issue, we treat the system-maintained benchmarks as a *context reservoir* and retrieve a local neighborhood $\mathcal{R}(x^*) := \mathcal{R}_{D^*}(x^*)$ from the anchored benchmark D^*. These semantically related, labeled instances enrich the query with task context and difficulty cues, yielding a more stable estimate of the capability direction required by x^*.

We then construct a query-side specification by applying the same parameter-vector extraction routine with the base model:

$$\boldsymbol{\tau}_{x^*} = \text{PAVE}\big(\mathcal{R}(x^*), \textsc{Normalize}(\{w_i'\}), \theta_0\big). \tag{6}$$

To minimize mismatch with offline construction, we reuse the baseline difficulty weights $\{w_i\}$ defined in Section 3.1.2 and only adjust them by semantic relevance:

$$w_i' = w_i \cdot \text{sim}\big(E(x^*), E(x_i)\big).$$

This preserves the original weighting scheme while focusing the query specification on samples most representative of x^*.

Matching with Confidence-Aware Aggregation. For each agent f, we match the query specification $\boldsymbol{\tau}_{x^*}$ against its hierarchical set $\mathcal{T}_f(D^*)$. To handle within-domain heterogeneity under limited query-side signal, we aggregate the top-k cosine similarities with decaying weights. If $\mathcal{T}_f(D^*) = \emptyset$, we omit agent f from matching. Otherwise, let $s_{f,1} \geq \cdots \geq s_{f,k}$ be the top-k similarities

Table 1: Accuracy (%) on in-system benchmarks. We construct specifications using the training split and evaluate on the held-out test split. The highest score among all methods for each benchmark is highlighted in bold.

Benchmark	GPT-4.1	Random	DescSim	DescLLM	Model-SAT	K-NN	EmbedLLM	Ours
GSM8K	94.30	78.40	74.40	72.90	94.60	92.60	96.30	**96.50**
MathQA	90.50	75.50	79.00	78.90	90.00	82.40	88.30	**91.20**
MATH	88.70	74.70	69.70	70.80	90.30	87.60	95.40	**96.20**
MedMCQA	78.80	65.30	68.20	75.20	75.00	71.50	77.40	**79.80**
PubMedQA	69.60	58.00	60.80	69.40	66.80	69.80	72.40	**74.80**
FPB	**81.03**	72.78	70.31	77.73	74.64	78.87	80.52	**81.03**
Overall Avg. (↑)	83.82	70.78	70.40	74.16	81.89	80.46	85.05	**86.59**
Average Rank (↓)	2.83	7.17	7.17	6.00	4.33	4.67	2.67	**1.00**

between $\boldsymbol{\tau}_{x^*}$ and vectors in $\mathcal{T}_f(D^*)$ and compute

$$s_f = \Big(\sum_{j=1}^{k} \omega_j s_{f,j}\Big) \Big/ \Big(\sum_{j=1}^{k} \omega_j\Big), \tag{7}$$

where $\omega_1 \geq \cdots \geq \omega_k > 0$ are fixed decaying weights (e.g., $\omega_j = \gamma^{j-1}$, $\gamma \in (0,1)$). When benchmark coverage for x^* is weak, similarity is less reliable, so we use baseline-hard competence. Let $g_f(D^*)$ be the number of baseline-hard samples solved by f during profiling. We then fuse normalized similarity and hard capability,

$$\mathrm{score}_f = \alpha\, \bar{s}_f + (1-\alpha)\, \bar{g}_f, \tag{8}$$

where $\bar{s}_f$ and $\bar{g}_f$ are min–max normalized over the remaining agents, and α increases with matching confidence (measured by $\max_f s_f$). Algorithm 3 summarizes the procedure.

4 Experiments

4.1 Experimental Setup

Agent System. We construct a system of $K = 136$ heterogeneous agents across two categories: (1) *API-based agents* (40 agents): directly invoking online services from multiple providers including OpenAI, Google Gemini, DeepSeek, Qwen family, and Doubao; (2) *Tool-augmented agents* (96 agents): equipping base models with external tools via three integration approaches, including MCP servers for standardized tool protocols, LangChain for community tools (e.g., ArXiv, Wikipedia, DuckDuckGo), and specialized function-based toolkits covering mathematical operations, finance utilities, and general reasoning. This diverse ecosystem reflects real-world scenarios where agents vary in general reasoning, mathematical computation, information retrieval, and tool execution capabilities. Detailed agent configurations are provided in Appendix C.1.

Benchmarks. We evaluate on benchmarks spanning three domains (mathematics, healthcare, and finance) with a clear separation between *in-system* and *out-of-system* settings to assess both identification accuracy and generalization capability.

For *in-system evaluation*, we use 6 benchmarks maintained by the system for agent profiling: GSM8K [6], MATH [9], and MathQA [1] for mathematics; MedMCQA [29] and PubMedQA [17] for healthcare; FPB [25] for finance. These benchmarks cover diverse reasoning patterns including multi-step mathematical reasoning, domain-specific knowledge retrieval, and sentiment classification. The *training split* is used for specification construction, while the *test split* is used for evaluation and unseen for the system.

For *out-of-system evaluation*, we select benchmarks that are *not* included in the system-maintained pool, testing generalization to unseen data distributions. We use 4 mathematics subsets and 6 healthcare subsets from MMLU [8], as well as 2 finance benchmarks (MA [42] and German [10]) from FinBench [41]. Full benchmark details are provided in Appendix C.2.

Contenders. We compare our approach against several representative methods, with implementation details in Appendix C.3:

- *GPT-4.1*: Directly answer user queries using GPT-4.1 without agent selection, serving as a strong single-model baseline.
- *Random*: Randomly select an agent to answer each query.
- *DescSim*: Select agents based on embedding similarity between user queries and agent descriptions using an embedding model.
- *DescLLM*: Use GPT-4o-mini to select agents based on queries and agent descriptions in a single prompt.
- *K-NN* [13, 35]: Retrieve nearest training samples and select the agent with the best average performance on these samples.
- *Model-SAT* [44]: A method that derives capability summaries from benchmark performance and uses a zero-shot LLM to estimate an agent's probability of correctly answering the query.
- *EmbedLLM* [47]: Learn compact agent embeddings via an encoder-decoder framework trained on all benchmark evaluation results, and use a linear router to combine agent and query embeddings to predict answer correctness, routing each query to the agent with the highest predicted probability.

Evaluation Protocol and Metrics. We report *agent identification accuracy*, which measures how often the selected agent solves the query. For each test sample (x, y), we (1) use the compared method to select one agent f, (2) invoke f to obtain the response $f(x)$, and (3) compute the quality $Q(f(x), y)$ against the reference answer y. The value $Q(f(x), y)$ is the per-sample score for that query (1 if correct, 0 otherwise in our benchmarks). The reported accuracy is the average of Q over all test samples in the benchmark. The quality function Q is task-specific: for mathematical reasoning, we extract the final numerical or symbolic answer and use normalized exact match; for multiple-choice tasks, we parse the selected option or option text; for FPB sentiment classification, we map the output to a sentiment label and compare it with the reference. The same Q is used during offline profiling to define successful samples ($Q \geq \delta$), so the evaluation metric is aligned with the signal used to build constructive specifications.

Table 2: Accuracy (%) on out-of-system benchmarks, excluded from the system-maintained collection and used for evaluating generalization to unseen tasks. The highest score among all methods for each benchmark is highlighted in bold.

Benchmark	GPT-4.1	Random	DescSim	DescLLM	Model-SAT	K-NN	EmbedLLM	Ours
MMLU (Mathematics, Avg)	94.23	83.58	82.56	91.12	90.47	90.34	97.33	**98.01**
- abstract_algebra	91.00	78.00	69.00	91.00	89.00	88.00	96.00	**97.00**
- college_mathematics	91.00	77.00	81.00	91.00	84.00	89.00	**97.00**	**97.00**
- elementary_mathematics	97.88	93.39	93.92	92.86	96.30	94.71	98.15	**98.41**
- high_school_mathematics	97.04	85.93	86.30	89.63	92.59	89.63	98.15	**99.63**
MMLU (Healthcare, Avg)	93.22	85.99	88.87	92.73	89.48	90.51	93.17	**93.69**
- anatomy	88.15	77.78	80.00	88.89	83.70	84.44	88.89	**91.85**
- clinical_knowledge	**90.94**	83.77	86.79	90.94	87.17	88.68	**90.94**	90.19
- college_biology	**98.61**	90.28	93.75	97.22	96.53	95.14	**98.61**	**98.61**
- college_medicine	87.28	83.24	87.86	86.71	82.08	**88.44**	**88.44**	87.28
- medical_genetics	98.00	93.00	94.00	97.00	94.00	93.00	98.00	**99.00**
- professional_medicine	**96.32**	87.87	90.81	95.59	93.38	93.38	94.12	95.22
Finance (Avg)	49.60	45.00	40.50	38.45	54.15	45.55	46.75	**67.90**
- MA	63.20	67.00	71.00	59.40	63.80	63.60	59.00	**83.80**
- German	36.00	23.00	10.00	17.50	44.50	27.50	34.50	**52.00**
Overall Avg. (↑)	86.29	78.36	78.70	83.15	83.92	82.96	86.82	**90.83**
Average Rank (↓)	2.80	7.07	6.40	4.73	4.87	5.07	2.60	**1.47**

Implementation Details. For specification construction, we use Qwen3-Embedding-0.6B for semantic embeddings and Qwen2.5-0.5B as the base model $h(\cdot;\theta_0)$ for PAVE generation. The profiling budget is $B = 300$ agent calls per benchmark. Complete hyperparameter settings are provided in Appendix C.3.

4.2 Main Results

Table 1 and Table 2 report agent identification accuracy on in-system and out-of-system benchmarks, respectively. Across the six in-system benchmarks, our approach attains the highest overall average accuracy (86.59%) with the best average rank (1.00), and is the top performer on all benchmarks (tied on FPB). On the out-of-system benchmarks, our method again yields the best overall average accuracy (90.83%) with the best average rank (1.47). While it is not the best method on every individual subset, it achieves strong and stable performance across domains, including a large margin on the finance benchmarks.

Comparison with Learning-Based Methods. Among learning-based contenders, EmbedLLM performs strongly on the in-system setting (85.05%), but our method is higher by 1.54% in average accuracy. More importantly, the advantage becomes larger on out-of-system benchmarks (90.83% vs. 86.82%, +4.01%). This pattern suggests that our specification-based design is more robust to distribution shift in the benchmark pool. A plausible reason is that we avoid fitting a single end-to-end router with benchmark-wide supervision, and instead enrich the query context with benchmark signals before matching to agents using multi-granularity specifications, and the ablation in Section 4.3 further supports the value of hierarchical structure.

Comparison with a Single Large Model. GPT-4.1 serves as a strong single-model baseline, achieving 83.82% accuracy on in-system and 86.29% on out-of-system benchmarks. Our method improves over GPT-4.1 by 2.77% and 4.54% in overall average accuracy on the two settings, respectively, while routing over a pool consisting only of lightweight agents. Overall, these results suggest that, on the evaluated benchmark suite, routing among specialized agents can match or exceed a single general-purpose model, as different agents may be better suited to different query types (e.g., tool use, domain-specific knowledge, or reasoning patterns). Meanwhile, this advantage is contingent on the composition and capability of the agent pool, and does not imply that routing will consistently outperform a stronger standalone model in all scenarios.

4.3 Ablation Study

We analyze how key components in our identification pipeline contribute to performance under both *in-system* and *out-of-system* benchmarks. Table 3 reports overall averaged accuracy and the absolute drop from the full method.

Fusion benefits from combining complementary signals. Our score fuses a similarity-based matching signal and a confidence-aware signal derived from demonstrated successes (Section 3.2). Removing either side degrades performance: *w/o* $\bar{s}_f$ *fusion* drops by 0.20 (in-system) and 0.78 (out-of-system), while *w/o* $\bar{g}_f$ *fusion* drops by 0.37 and 1.55 points, respectively. These results justify fusing the two signals: similarity measures how well a query matches an agent in the shared space, while the confidence-aware term provides an additional reliability cue from the profiled success set. Notably, the ablation gaps are consistently larger on out-of-system benchmarks, suggesting that confidence-aware aggregation and similarity fusion contribute to more robust out-of-system generalization.

Hierarchical specification captures heterogeneity. Replacing hierarchical specifications with a non-hierarchical alternative (*w/o hierarchy*) leads to a consistent degradation (0.97 / 1.12 drop on in-/out-of-system). This is in line with the intended role of hierarchy: preserving cluster-level structure can retain fine-grained capability cues that may be diluted by a single global representation. The slightly larger drop on out-of-system further suggests that

Table 3: Ablation summary on in-/out-of-system benchmarks. We report Acc. (%) and absolute drop vs. Full (Drop, %).

Method	In-System	Drop	Out-of-System	Drop
Full (Ours)	**86.59**	–	**90.83**	–
w/o $\bar{s}_f$ fusion	86.39	0.20	90.05	0.78
w/o $\bar{g}_f$ fusion	86.22	0.37	89.28	1.55
w/o hierarchy	85.62	0.97	89.71	1.12
w/o active prof.	85.88	0.71	88.88	1.95

such structure can be beneficial under distribution shifts, without implying it is the only factor behind generalization.

Active profiling improves specification quality. When we disable active profiling and build specifications from randomly sampled instances under the same budget (*w/o active prof.*), accuracy decreases by 0.71 in-system and 1.95 out-of-system. This suggests that active exploration helps allocate evaluations to more informative regions, yielding specifications that are more discriminative for downstream matching. The larger gap on out-of-system benchmarks is consistent with the intuition that active profiling tends to matter more when the evaluation distribution differs from the system-maintained benchmarks.

4.4 Analysis

Specification Size and Storage. Each agent's specification is a set of hierarchical PAVE vectors stored as LoRA-style low-rank updates. Across 136 agents and 6 system-maintained benchmarks, an agent has 2.5 benchmark-level specifications on average (ranging from 0 to 6), and each specification contains 1 global vector plus 10.3 cluster vectors on average. Figure 2 (left) shows that the resulting per-agent storage is 49.0 MB on average (ranging from 0.26 to 111.92 MB), which is comparable to storing a small model checkpoint and is practical to cache and distribute in plug-and-play settings (specifications are read-only at serving time and can be fetched on demand). Moreover, Figure 2 (right) confirms near-linear scaling with the number of maintained benchmarks ($y = 15.74x + 9.28$, $R^2 = 0.935$), making the storage growth predictable as the system-maintained benchmark collection expands.

Importantly, this storage cost can be reduced without changing the method. First, we can limit hierarchical granularity by retaining fewer clusters per benchmark (e.g., only those with sufficient successes), directly reducing stored vectors. Second, we can shrink the update space by applying LoRA to fewer reference-model modules/layers (or using a smaller rank), producing smaller low-rank updates while preserving the shared matching space. Together, these knobs yield a clear accuracy–storage trade-off and make the footprint configurable to deployment budgets.

Computational Overhead. Specification construction is performed once per agent. For each agent–benchmark pair, active profiling uses a *dynamic* budget of at most $B = 300$ agent calls: profiling can terminate early once the success set is sufficient or the agent is unlikely to improve. Wall-clock time depends on the agent's API or local inference latency, on the order of several minutes to tens of minutes per pair with typical API-based models when the full budget is used. PAVE extraction and hierarchical clustering are run on the profiled success set using the lightweight base

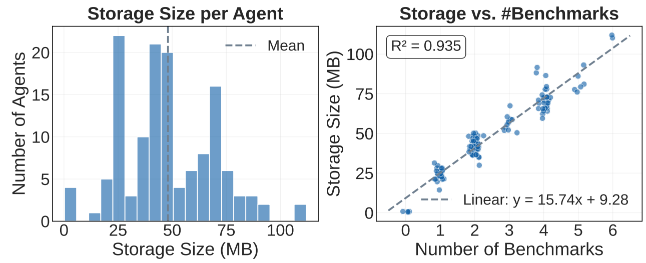

Figure 2: Specification size and storage (left: per-agent distribution; right: storage vs. benchmark count).

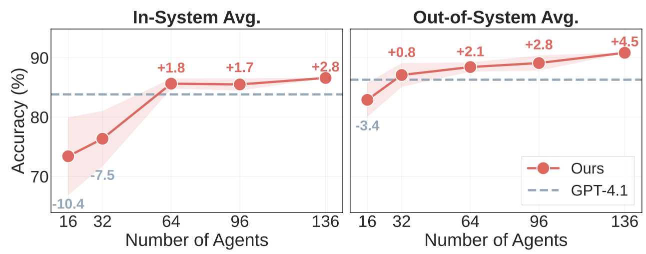

Figure 3: Scalability w.r.t. agent system size: in-/out-of-system average accuracy (shaded: std over 3 random subsets from the full system), with annotated gains over GPT-4.1.

model (Qwen2.5-0.5B) and add negligible cost relative to profiling. Building full specifications for all 136 agents over 6 benchmarks is feasible on a single machine over a few hours to a day, depending on agent response times.

At serving time, given a query we: (1) anchor to a benchmark and form the query specification via retrieved neighbors, and (2) compute similarity between the query specification and each agent's hierarchical specifications. In our prototype, benchmark anchoring and query specification take under 3 seconds, and similarity matching takes under 0.1 seconds. Since end-to-end routing latency is dominated by the selected agent's response time, our selection overhead remains a small fraction of the user-visible delay.

The current numbers leave substantial room for acceleration. For benchmark anchoring and neighbor retrieval, lighter retrieval models, approximate nearest-neighbor search, and parallel retrieval can reduce cost without changing the method. For query-specification construction, one can use a lighter or quantized base model, efficient PEFT variants, fewer retrieved neighbors, and caching for similar queries. For specification matching, parallel similarity computation or vector indexing can further accelerate scoring while preserving the same selection rule.

Scalability. Our plug-and-play design scales naturally with system size: adding agents only requires constructing specifications offline, without retraining any centralized selector. Figure 3 varies the number of available agents from 16 to 136. As K increases, identification accuracy improves and is stable across random seeds: in-system accuracy shows a large gain when moving from small to medium pools and remains consistently high for larger K, while out-of-system accuracy continues to increase and reaches 90.83% at

K=136. Moreover, the advantage over GPT-4.1 widens with larger system size, reaching +2.8 (in-system) and +4.5 (out-of-system) at K=136, confirming robust specification matching at scale.

5 Related Work

Learnware Specifications. The learnware paradigm [45, 46] envisions an open *dock system* that hosts reusable models together with compact *specifications*, enabling privacy-preserving identification and reuse without exposing raw training data [19]. The Beimingwu[1] platform [37] instantiates such a system, supporting active model management beyond hosting. Classic learnware specifications for conventional ML often summarize training data distributions via RKME (reduced kernel mean embedding) [40], and support identification via specification matching in a learnware system [22, 23, 36]. For language-model learnwares, recent work further explores PAVE (parameter vector) specifications [32, 38] to represent model capabilities in a shared parameter space, conceptually related to *task vectors* that view task adaptation as directions in weight space [14, 28]. Our *constructive specification* extends this line to *black-box agents* with inaccessible training data and internals. By budgeted active profiling on organized benchmarks, it constructs hierarchical PAVE specifications and supports *query-level* identification with benchmark-anchored query enrichment.

Model Selection and Routing. In addition to learnware, model selection and routing typically follow two lines: (i) *description-based* dispatch using semantic specifications, and (ii) *router-based* selection learned from supervision over a fixed candidate pool. HuggingGPT [31], for instance, selects models based on their function descriptions. A second line learns or designs selectors that choose which model to invoke for each query: LLM-Blender [15] combines outputs from multiple models, while FrugalGPT [5] uses cascade-style policies that adaptively decide whether to invoke additional models. Other work trains a dedicated router using labeled or preference supervision over a target pool, as in RouteLLM [26]. Retrieval-style baselines retrieve the nearest samples and select the model with the best average performance on those neighbors [13, 35]. More recent proposals learn compact *model embeddings* from benchmark evaluation matrices and train lightweight predictors (EmbedLLM) [47], or derive *capability summaries* from benchmark performance and prompt a lightweight LLM to estimate per-query success probabilities (Model-SAT) [44]. However, description-based selection can be coarse and may deviate from actual capabilities, while router-based methods are pool-dependent: adding new candidates typically requires new comparisons and supervision, followed by router retraining or recalibration as the system evolves. In contrast, our approach removes both description dependence and centralized router training: it constructs agent capability specifications offline via budgeted active profiling, and performs online selection by similarity matching in a unified specification space, enabling *plug-and-play, query-level* identification in evolving ecosystems. Importantly, this identification problem becomes even more central when moving from *one-step model routing* to *multi-step agentic workflows*, where systems must repeatedly select and re-select tools/agents along the trajectory.

[1]https://github.com/Learnware-LAMDA

LLM Agents and Multi-Agent Systems. As LLM systems become increasingly *agentic*, the selection problem shifts from *single-shot routing* to *repeated, step-wise orchestration*: an agent must dynamically choose tools, sub-agents, or specialized components along a multi-step workflow, while the underlying component pool can continually evolve. LLM-powered agents address real-world tasks via planning and tool invocation [21, 24], and methods such as ReAct [43] and Reflexion [34] improve tool use and iterative refinement. Tool-use capabilities are advanced by approaches such as Toolformer [30] and Search-R1 [16], which train language models to decide when and how to invoke tools during inference. Meanwhile, multi-agent frameworks [11, 20] increase orchestration complexity by coordinating specialized agents to tackle complex tasks collaboratively. In such settings, relying on textual descriptions or pool-dependent routers becomes increasingly brittle, as new agents/tools may appear without training data or supervision for re-training. Complementing efforts on making agents stronger and better coordinated, we focus on the *system problem* of *plug-and-play, query-level identification* in an evolving pool of *black-box* agents, and propose a benchmark-driven constructive specification mechanism that supports accurate, budget-efficient selection without relying on descriptions or retraining routers.

6 Conclusion

This paper studies *query-level* agent identification in evolving learnware systems, where candidates are black-box services with no accessible internals or training data, and exhaustive evaluation or router retraining is impractical. To address this challenge, we propose *constructive specification*: each agent is profiled on system-maintained benchmarks under a strict budget, demonstrated successes are encoded as hierarchical parameter-vector specifications in a unified space, and each incoming query is mapped into the same space for similarity-based selection.

Our design is *plug-and-play*: new agents are added by computing specifications offline, and removing agents does not require updating any global selector. Across diverse evaluation scenarios, including out-of-distribution settings, experiments show that constructive specifications enable accurate and robust identification relative to representative baselines.

Future work includes extending agent identification beyond single-turn queries to multi-step workflows that require selecting and re-selecting agents across turns, and incorporating latency and compute cost into the objective for resource-aware identification under practical deployment constraints at scale.

Acknowledgments

This work was supported by 111 Center (B26023), Jiangsu Science Foundation Leading-edge Technology Program (BK20232003), and FIDBP of MoE (JYB2025XDXM118). Zhi-Hao Tan was supported by the Postdoctoral Fellowship Program of CPSF (GZB20250396) and Jiangsu Funding Program for Excellent Postdoctoral Talent (2025ZB482). This work was also supported by Nanjing University-China Mobile Communications Group Co., Ltd. Joint Institute. We thank Jia-Wei Shan and Peng Tan for their helpful discussions. We are also grateful to the anonymous reviewers for their constructive comments and valuable suggestions.

References

[1] Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms. arXiv:1905.13319

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2-3 (2002), 235–256.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33*. 1877–1901.

[4] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. 2011. X-Armed Bandits. *Journal of Machine Learning Research* 12, 5 (2011), 1655–1695.

[5] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. arXiv:2305.05176

[6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. arXiv:2110.14168

[7] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081 (2025), 633–638.

[8] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *Proceedings of the 9th International Conference on Learning Representations*.

[9] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. arXiv:2103.03874

[10] Hans Hofmann. 1994. Statlog (German Credit Data). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5NC77.

[11] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *Proceedings of the 12th International Conference on Learning Representations*.

[12] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *Proceedings of the 10th International Conference on Learning Representations*.

[13] Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. 2024. RouterBench: A benchmark for multi-LLM routing system. arXiv:2403.12031

[14] Gabriel Ilharco, Marco Túlio Ribeiro, Mitchell Wortsman, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. 2023. Editing models with task arithmetic. In *Proceedings of the 11th International Conference on Learning Representations*.

[15] Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. LLM-Blender: Ensembling Large Language Models with Pairwise Ranking and Generative Fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. 14165–14178.

[16] Bowen Jin, Hansi Zeng, Zhenrui Yue, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning. arXiv:2503.09516

[17] Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William Cohen, and Xinghua Lu. 2019. PubMedQA: A Dataset for Biomedical Research Question Answering. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing*. 2567–2577.

[18] Robert Kleinberg, Aleksandrs Slivkins, and Eli Upfal. 2008. Multi-armed bandits in metric spaces. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*. 681–690.

[19] Hao-Yi Lei, Zhi-Hao Tan, and Zhi-Hua Zhou. 2024. On the ability of developers' training data preservation of learnware. In *Advances in Neural Information Processing Systems 37*. 36471–36513.

[20] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. In *Advances in Neural Information Processing Systems 36*. 51991–52008.

[21] Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, et al. 2025. Deepagent: A general reasoning agent with scalable toolsets. arXiv:2510.21618

[22] Jian-Dong Liu, Zhi-Hao Tan, and Zhi-Hua Zhou. 2024. Towards making learnware specification and market evolvable. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*. 13909–13917.

[23] Jian-Dong Liu, Zhi-Hao Tan, and Zhi-Hua Zhou. 2025. Identifying and Reusing Learnwares Across Different Label Spaces. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence*. 5734–5742.

[24] Pan Lu, Bowen Chen, Sheng Liu, Rahul Thapa, Joseph Boen, and James Zou. 2025. Octotools: An agentic framework with extensible tools for complex reasoning. arXiv:2502.11271

[25] Pekka Malo, Ankur Sinha, Pekka Korhonen, Jyrki Wallenius, and Pyry Takala. 2014. Good debt or bad debt: Detecting semantic orientations in economic texts. *Journal of the Association for Information Science and Technology* 65, 4 (2014), 782–796.

[26] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. 2025. RouteLLM: Learning to Route LLMs from Preference Data. In *Proceedings of the 13th International Conference on Learning Representations*.

[27] OpenAI. 2023. GPT-4 technical report. arXiv:2303.08774

[28] Guillermo Ortiz-Jimenez, Alessandro Favero, and Pascal Frossard. 2023. Task arithmetic in the tangent space: Improved editing of pre-trained models. In *Advances in Neural Information Processing Systems 36*. 66727–66754.

[29] Ankit Pal, Logesh Kumar Umapathi, and Malaikannan Sankarasubbu. 2022. MedMCQA: A Large-scale Multi-Subject Multi-Choice Dataset for Medical domain Question Answering. In *Proceedings of the Conference on Health, Inference, and Learning*. 248–260.

[30] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36*. 68539–68551.

[31] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In *Advances in Neural Information Processing Systems 36*. 38154–38180.

[32] Hao-Yu Shi, Zhi-Hao Tan, Zi-Chen Zhao, Yang Yu, and Zhi-Hua Zhou. 2026. A Study on PAVE Specification for Learnware. In *Proceedings of the 14th International Conference on Learning Representations*.

[33] Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. 2025. Retrieval Models Aren't Tool-Savvy: Benchmarking Tool Retrieval for Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2025*. 24497–24524.

[34] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36*. 8634–8652.

[35] Tal Shnitzer, Anthony Ou, Mírian Silva, Kate Soule, Yuekai Sun, Justin Solomon, Neil Thompson, and Mikhail Yurochkin. 2023. Large Language Model Routing with Benchmark Datasets. arXiv:2309.15789

[36] Peng Tan, Hai-Tian Liu, Zhi-Hao Tan, and Zhi-Hua Zhou. 2024. Handling learnwares from heterogeneous feature spaces with explicit label exploitation. In *Advances in Neural Information Processing Systems 37*. 12767–12795.

[37] Zhi-Hao Tan, Jian-Dong Liu, Xiao-Dong Bi, Peng Tan, Qin-Cheng Zheng, Hai-Tian Liu, Yi Xie, Xiao-Chuan Zou, Yang Yu, and Zhi-Hua Zhou. 2024. Beimingwu: A learnware dock system. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5773–5782.

[38] Zhi-Hao Tan, Zi-Chen Zhao, Hao-Yu Shi, Xin-Yu Zhang, Peng Tan, Yang Yu, and Zhi-Hua Zhou. 2025. Learnware of language models: Specialized small language models can do big. arXiv:2505.13425

[39] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[40] Xi-Zhu Wu, Wenkai Xu, Song Liu, and Zhi-Hua Zhou. 2023. Model reuse with reduced kernel mean embedding specification. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2023), 699–710.

[41] Qianqian Xie, Weiguang Han, Zhengyu Chen, Ruoyu Xiang, Xiao Zhang, Yueru He, Mengxi Xiao, Dong Li, Yongfu Dai, Duanyu Feng, et al. 2024. The FinBen: A Holistic Financial Benchmark for Large Language Models. arXiv:2402.12659

[42] Linyi Yang, Eoin Kenny, Tin Lok James Ng, Yi Yang, Barry Smyth, and Ruihai Dong. 2020. Generating plausible counterfactual explanations for deep transformers in financial text classification. In *Proceedings of the 28th international conference on computational linguistics*. 6150–6160.

[43] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations*.

[44] Yi-Kai Zhang, De-Chuan Zhan, and Han-Jia Ye. 2025. Capability Instruction Tuning. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence*. 25958–25966.

[45] Zhi-Hua Zhou. 2016. Learnware: on the future of machine learning. *Frontiers of Computer Science* 10, 4 (2016), 589–590.

[46] Zhi-Hua Zhou and Zhi-Hao Tan. 2024. Learnware: Small models do big. *Science China Information Sciences* 67, 1 (2024), 112102.

[47] Richard Zhuang, Tianhao Wu, Zhaojin Wen, Andrew Li, Jiantao Jiao, and Kannan Ramchandran. 2025. EmbedLLM: Learning Compact Representations of Large Language Models. In *Proceedings of the 13th International Conference on Learning Representations*.

Supplementary Materials

The appendix is organized as follows: notation summary, proofs of theoretical results, and additional experimental details.

A Notations

The major notations used in this work are summarized in Table 4.

Table 4: Major notations used in this work.

Category	Notation	Description
Learnware Dock System	$\mathcal{F} = \{f_1, \ldots, f_K\}$	The collection of K agents managed by the learnware dock system.
	f_k	A black-box agent that processes query x and produces a response $f_k(x)$.
	$\mathcal{D} = \{D_1, \ldots, D_M\}$	The set of M reference benchmarks maintained by the system for capability profiling.
	$D_m = \{(x_i, y_i)\}_{i=1}^{N_m}$	The m-th benchmark consisting of N_m query-answer pairs from a specific domain.
	$E : \mathcal{X} \to \mathbb{R}^d$	Embedding function mapping queries to d-dimensional vectors.
	$Q : \mathcal{Y} \times \mathcal{Y} \to [0, 1]$	Quality function measuring the quality of a response against a reference answer.
Constructive Specification	$\mathcal{P}_f(D)$	Performance profile of agent f on benchmark, containing samples where $Q(f(x), y) \geq \delta$.
	δ	Quality threshold that determines whether an agent successfully handles a given sample.
	$h(\cdot; \theta_0)$	Pre-trained base model as the base for computing parameter vectors.
	$\tau_f(D)$	Parameter vector specification fitted on the profile $\mathcal{P}_f(D)$ w.r.t. the base model.
	$\mathcal{T}_f(D)$	Hierarchical specification set containing global and cluster-specific specifications.
	$\mathcal{S}_f(\mathcal{D})$	Agent-level constructive specification over the benchmark collection $\mathcal{D}$.
	$n_{\min}$	Minimum profile size for a reliable specification, ensuring stable parameter vector fitting.
	C	Number of clusters for partitioning the performance profile into sub-groups.
Capability Profiling	$u_f(x)$	Performance function of agent f on sample x, defined as $Q(f(x), y)$.
	B	Query budget limiting the number of agent invocations during capability profiling.
	$U_f(x')$	Optimistic estimate for an unevaluated sample x' induced by the evaluated set $\mathcal{V}$.
Agent Search	x^*	User query submitted to the agent system for specification-based agent selection.
	D^*	Anchored benchmark selected as the most relevant to the user query x^*.
	τ_{x^*}	Query specification constructed from x^* via retrieved neighborhood.
	s_f	Similarity score between τ_{x^*} and agent f's hierarchical specifications.
	score_f	Final ranking score fusing similarity s_f and baseline-hard competence.

B Proofs

This section provides proofs for the theoretical results in Section 3.1.

B.1 Proof of Theorem 3.3

Proof. Recall the optimistic upper bound:

$$U_f(x') = \min_{x \in \mathcal{V}} \Big(u_f(x) + \beta \cdot d(x', x) \Big).$$

For any $x \in \mathcal{V}$, the L-Lipschitz condition gives

$$u_f(x') - u_f(x) \leq |u_f(x') - u_f(x)| \leq L \cdot d(x', x).$$

Rearranging: $u_f(x') \leq u_f(x) + L \cdot d(x', x)$. Since $\beta \geq L$, we have

$$u_f(x') \leq u_f(x) + \beta \cdot d(x', x).$$

This holds for all $x \in \mathcal{V}$. Taking the minimum, we have:

$$u_f(x') \leq \min_{x \in \mathcal{V}} \Big(u_f(x) + \beta \cdot d(x', x) \Big) = U_f(x'),$$

which completes the proof. □

B.2 Proof of Corollary 3.4

Proof. Whenever Algorithm 1 updates $U^*(x')$, it sets

$$U^*(x') \leftarrow \min\{U^*(x'),\ u_f(x) + \beta d(x', x)\}$$

for $x \in \mathcal{V}$. By Theorem 3.3, for every $x \in \mathcal{V}$ we have

$$u_f(x') \leq u_f(x) + \beta d(x', x).$$

Since $U^*(x')$ is the running minimum of such values, we obtain $u_f(x') \leq U^*(x')$. Therefore, if $U^*(x') < \delta$, then $u_f(x') < \delta$. □

B.3 Proof of Proposition 3.5

Proof. Each evaluation adds one new element into $\mathcal{V}$, so $|\mathcal{V}| \leq B$. A heap push occurs only after a successful tightening $U_{\text{new}} < U^*(x')$ inside the n-neighbor loop, hence each evaluation triggers at most n pushes. Therefore,

$$\#\text{Push} \leq k + n|\mathcal{V}| \leq k + nB.$$

Since every pop removes one item, the total number of pops cannot exceed the total number of pushes:

$$\#\text{PopMax} \leq \#\text{Push} \leq k + nB.$$

With a binary heap, the total time complexity is

$$O\big((\#\text{Push} + \#\text{PopMax}) \log(k + nB)\big),$$

which is equivalent to $O\big((k + nB) \log(k + nB)\big)$. □

C Additional Experimental Details

This section provides the additional experimental details.

C.1 Agent System

Our agent system comprises $K = 136$ heterogeneous agents spanning two categories: API-based agents and tool-augmented agents. Below we provide detailed descriptions of each category along with their construction processes.

API-Based Agents. We include API-accessible lightweight language models from multiple providers: OpenAI (gpt-3.5-turbo, gpt-4o-mini, gpt-4.1-nano, gpt-4.1-mini), Google (gemini-2.5-flash), DeepSeek (deepseek-v3.2, deepseek-v3, deepseek-r1-distill-qwen-7b), Qwen (qwen-turbo, qwen-flash, qwen-math-plus, qwen-long, qwen-coder-turbo, qwen-coder-plus, qwen-max, qwen-mt-flash, qwen-plus, qwen2-7b-instruct, qwen2.5-7b-instruct, qwen2.5-7b-instruct-1m, qwen2.5-14b-instruct-1m, qwen2.5-32b-instruct, qwen2.5-coder-7b-instruct, qwen2.5-coder-14b-instruct, qwen2.5-coder-32b-instruct, qwen2.5-7b-instruct, qwen2.5-3b-instruct, qwen3-coder-flash, qwen3-coder-30b-a3b-instruct, qwen3-next-80b-a3b-instruct, qwen3-30b-a3b-instruct, qwen3-30b-a3b-thinking, qwen3-32b, qwen3-14b, qwen3-8b, qwen3-4b, qwen3-1.7b, qwen3-0.6b), and Doubao (doubao-1-5-lite-32k, doubao-seed-1-6-flash). At inference time, we call the model's chat completion API with the user query and return the generated response.

Tool-Augmented Agents. We construct tool-augmented agents using three approaches, including MCP-based agents, function-based agents, and LangChain-based agents:

- *MCP-based Agents:* Using the CAMEL [20] framework, we equip base models (GPT-4.1-mini, GPT-4o-mini, GPT-4.1-nano) with 14 MCP tool packages: zhipu-web-search-sse, calculator, deephub-wiki, bilibili-search, trends-hub, think-tool, sequential-thinking, math-tools, math-mcp, mcp-math-eval, calculator-mcp, code-reasoning, claude-code-mcp, and codegen-mcp. We deploy an MCP server, load the server configuration, and initialize the underlying model. During inference, the agent executes tool calls via an MCP client and returns the final response.
- *Function-based Agents:* We integrate base models with a curated set of functions from the ToolRet [33] dataset, covering 13 thematic tool groups: basic math, number theory, algebra, linear algebra, geometry, unit conversion, statistics, sequences, strings and digits, set counting, finance, time/date, and general utilities. During inferencing, the agent selects and invokes tools as needed and returns the final answer.
- *LangChain-based Agents:* We equip base models with LangChain-community tools (arxiv, duckduckgo, wikipedia, google-news, wikidata). We first instantiate the specific tool and create the LLM and agent executor. Inference is carried out by invoking the agent executor with the user query, which internally manages tool calls and response generation.

C.2 Benchmarks

C.2.1 In-System Benchmarks. We use six benchmarks across mathematics, healthcare, and finance. Their training splits support profiling and specification construction, while their test splits evaluate in-system performance.

- *GSM8K* [6]: 8.5K high-quality grade-school math word problems, with 7,473 training and 1,319 test samples.
- *MATH* [9]: 12,500 challenging problems from high school mathematics competitions, covering algebra, geometry, number theory, and calculus, with 7,500 training and 5,000 test samples.
- *MathQA* [1]: natural-language math problems paired with structured equation representations and final answers, with 29,837 training and 2,985 test samples.
- *MedMCQA* [29]: medical multiple-choice QA from entrance examinations, covering 21 subjects and over 2.4K topics, with 182,822 training and 6,150 test samples.
- *PubMedQA* [17]: biomedical QA over PubMed abstracts, with 450 training and 500 test samples.
- *FPB* [25]: financial sentiment classification over news and report sentences, with 3,100 training and 970 test samples.

C.2.2 Out-of-System Benchmarks. To evaluate generalization, we use held-out benchmarks from the same three domains but with different distributions and task characteristics. None are used for profiling or specification construction.

Mathematics Domain. We use four mathematics-related multiple-choice subsets from MMLU [8]:

- *Abstract Algebra* (100 samples): Groups, rings, and fields.
- *College Mathematics* (100 samples): Covers undergraduate-level topics in calculus, linear algebra, and differential equations.
- *Elementary Mathematics* (378 samples): Tests basic mathematical skills in arithmetic, fractions, and simple algebraic reasoning.
- *High School Mathematics* (270 samples): Evaluates high school level mathematics including algebra, geometry, and pre-calculus.

Healthcare Domain. We use six MMLU healthcare subsets:

- *Anatomy* (135 samples): Tests knowledge of human anatomical structures and systems.
- *Clinical Knowledge* (265 samples): Evaluates practical clinical reasoning and medical decision-making.
- *College Biology* (144 samples): Undergraduate biology.
- *College Medicine* (173 samples): Medical-school knowledge.
- *Medical Genetics* (100 samples): Genetics in medical contexts.
- *Professional Medicine* (272 samples): Evaluates knowledge required for medical licensing and professional practice.

Finance Domain. We select two financial benchmarks with distinct task types from FinBench [41]:

- *MA* [42] (500 samples): Classify whether an M&A deal in news or tweets was completed or remained a rumor.
- *German Credit* [10] (200 samples): Classify individuals as "good" or "bad" credit risks based on historical customer attributes.

C.2.3 Test Set Size. Due to computational constraints, we randomly sample 1,000 instances (with seed 42) from four benchmarks with large test sets: *GSM8K*, *MATH*, *MathQA*, and *MedMCQA*. All other benchmarks use their complete test sets.

C.3 Method Implementations

Contenders. We implement contenders as follows:

- *DescSim*: We construct agent descriptions by pairing model names with capability summaries, computing query–agent similarity via Qwen3-Embedding-0.6B.
- *DescLLM*: We include the descriptions of all agents in a single prompt and ask GPT-4o-mini to select the most suitable agent.
- *K-NN* [13, 35]: We retrieve the $k = 16$ most similar training queries and select the agent with the best average performance.
- *EmbedLLM* [47]: We learn 232-dimensional agent embeddings from the query–agent performance matrix.
- *Model-SAT* [44]: We summarize per-domain accuracies into textual agent capability representations and concatenate them with the user query and a performance inquiry prompt to construct capability instructions. These instructions are fed into Qwen3-8B in a zero-shot setting to predict the probability of each agent answering the query correctly.

Our Method. We use Qwen3-Embedding-0.6B for sample semantic embedding and Qwen2.5-0.5B as the base model for PAVE generation. Key hyperparameters: profiling budget $B = 300$ agent calls per benchmark, quality threshold $\delta = 0.7$, exploration coefficient $\beta = 0.2$, and neighborhood size $n = 10$. For hierarchical specification, we set maximum clusters $C_{\max} = 20$, minimum cluster size $n_{\min} = 16$, and sample weights $w_{\text{hard}} = 1.0$, $w_{\text{easy}} = 0.2$. During online search, we use top-k aggregation with $k = 3$ and decay weights $(0.6, 0.3, 0.1)$. All experiments use a fixed random seed of 42 for reproducibility.