CONSTRUCTIVE SPECIFICATION FOR PLUG-AND-PLAY LEARNWARE AGENTS

Jian-Dong Liu[1,2], **Zi-Chen Zhao**[1,2], **Hao Sun**[3], **Lin-Xing Wu**[1,2], **Huan Zhang**[3], **Pengyuan Wang**[1,2], **Ming Zhao**[3], **Xinyu Chu**[1,2], **Shu Yan**[1,2], **Yongbei Zhu**[3], **Weijun Zhong**[3], **Zhi-Hao Tan**[1,2], **Jing Shang**[3], **Yang Yu**[1,2], **Zhi-Hua Zhou**[1,2,*]

[1]National Key Laboratory for Novel Software Technology, Nanjing University, China
[2]School of Artificial Intelligence, Nanjing University, China
[3]China Mobile Information Technology Center, 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.

1 INTRODUCTION

The recent surge of large language models (LLMs) (Brown et al., 2020; OpenAI, 2023; Guo et al., 2025) and their rapid deployment as API-accessible agents (Yao et al., 2022; Wang et al., 2024; Jin et al., 2025) 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* (Zhou, 2016; Zhou & Tan, 2024) 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) (Tan et al., 2024b), 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.

*Corresponding author (email: zhouzh@lamda.nju.edu.cn)

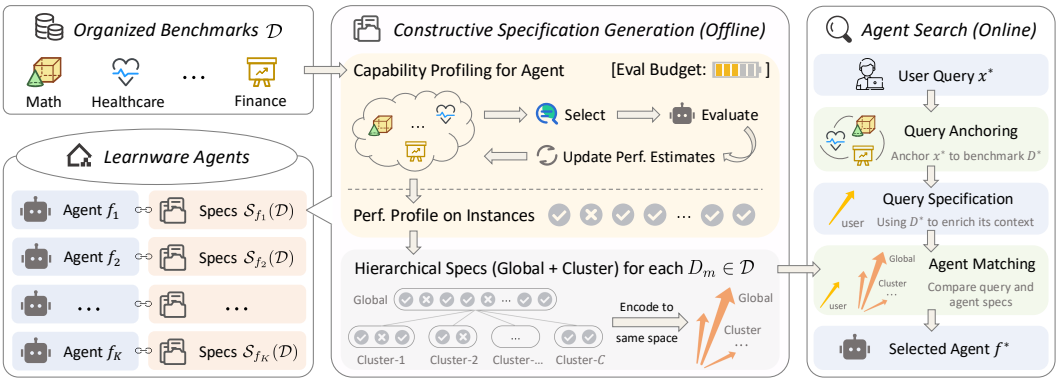

Figure 1: System overview of constructive specification for plug-and-play learnware agents.

Prior learnware work has demonstrated effectiveness on *task-level* reuse (Tan et al., 2024a; Liu et al., 2024; Tan et al., 2025), 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 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 (Schick et al., 2023; Shen et al., 2023) underpin semantic specifications, yet such descriptions are often too coarse and can diverge from actual capabilities. Router-based approaches train a dedicated selector (Zhuang et al., 2025; Ong et al., 2025) 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* (Tan et al., 2025; Shi et al., 2026), 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 underspecified, 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 space. Queries are projected into this space for multi-granularity matching, enabling plug-and-play identification as the system 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.
- **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 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 requiring model internals (e.g., weight inspection or gradient 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 rest.

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 benchmarks, 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 question becomes how to allocate this limited budget to obtain effective capability representations for 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 reflects stochasticity in agent responses. This setting is 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. equation 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. equation 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* (Auer et al., 2002): 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 $\mathcal{V}$:

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

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 (Kleinberg et al., 2008; Bubeck et al., 2011): 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). The complete procedure is summarized in Algorithm 1.

Theoretical Analysis. We analyze Algorithm 1 from the perspective of *Lipschitz/metric bandits* (Kleinberg et al., 2008; Bubeck et al., 2011): 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 C.

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 $x' \in D$,*

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

Theorem 3.3 shows that $U_f(\cdot)$ is a valid optimistic envelope: the true performance never exceeds its optimistic estimate. Based on this, Corollary C.1 establishes the safety of threshold pruning, and Proposition C.2 shows the lazy priority queue keeps overhead at $\mathcal{O}((k + nB)\log(k + nB))$. Both are stated and proved in Appendix C.

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 (Tan et al., 2025; Shi et al., 2026). 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:

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

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 (Hu et al., 2022) manner via low-rank updates (Shi et al., 2026). 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. equation 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:

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

where $\mathcal{C}_f = \{c : |\mathcal{P}_f(D, c)| \geq n_{\min}\}$ 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 $C = \min(C_{\max}, \lfloor |\mathcal{P}_f(D)|/n_{\min} \rfloor)$. The complete generation process is summarized in Algorithm 2.

Repeating this process for each $D \in \mathcal{D}$ yields the agent-level specification $\mathcal{S}_f(\mathcal{D})$ defined in Eq. equation 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 specification space and perform similarity-based selection. 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:

$$\begin{aligned} \mathrm{rel}(D_m, x^*) &= \frac{1}{m} \sum\nolimits_{(x,y)\in\mathcal{R}_{D_m}(x^*)} \mathrm{sim}(E(x^*), E(x)) \,, \\ D^* &= \arg\max\nolimits_{D_m\in\mathcal{D}} \ \mathrm{rel}(D_m, x^*). \end{aligned} \tag{5}$$

Averaging over multiple neighbors reduces sensitivity to noise.

Query Specification. A single query x^* is typically short and under-specified, making direct projection into the specification space unstable and unreliable. We therefore 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:

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

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:

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

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

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

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 $\max_f s_f$). The complete search procedure is summarized in Algorithm 3.

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

Table 1: Accuracy (%) on in-system benchmarks. Specifications are built on training and evaluated on held-out test splits. Bold indicates the highest score among all methods for each benchmark.

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**

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 D.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 (Cobbe et al., 2021), MATH (Hendrycks et al., 2021b), and MathQA (Amini et al., 2019) for mathematics; MedMCQA (Pal et al., 2022) and PubMedQA (Jin et al., 2019) for healthcare; FPB (Malo et al., 2014) 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 (Hendrycks et al., 2021a), as well as 2 finance benchmarks (MA (Yang et al., 2020) and German (Hofmann, 1994)) from FinBench (Xie et al., 2024). Full benchmark details are provided in Appendix D.2.

Contenders. We compare with several representative methods (details in Appendix D.6):

- *GPT-4.1*: A strong single-model baseline using GPT-4.1 without agent selection.
- *Random*: Randomly select an agent to answer each query.
- *DescSim*: Agent selection based on description-query similarity via an embedding model.
- *DescLLM*: Use GPT-4o-mini to select agents based on queries and descriptions in a single prompt.
- *K-NN* (Shnitzer et al., 2023; Hu et al., 2024): Retrieve nearest training samples and select the agent with the best average performance on these samples.
- *Model-SAT* (Zhang et al., 2025): Use zero-shot LLMs to estimate agent success probabilities from capability summaries on benchmarks.
- *EmbedLLM* (Zhuang et al., 2025): Learns joint agent-query embeddings to predict and route.

Implementation Details. We employ Qwen3-Embedding-0.6B and Qwen2.5-0.5B for embedding and PAVE generation ($h(\cdot;\theta_0)$), respectively. With a profiling budget of $B = 300$ per benchmark, additional details are in Appendix D.7.

4.2 Main Results

Across six in-system benchmarks, our approach achieves the highest average accuracy (86.59%) and a perfect average rank of 1.00 (Table 1). On out-of-system benchmarks, our method remains the

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**

top performer with 90.83% accuracy and an average rank of 1.47 (Table 2), demonstrating strong stability across domains, particularly in finance.

Comparison with Learning-Based Methods. Our method outperforms the strongest learning-based baseline, EmbedLLM, by 1.54% in-system and by a larger margin of 4.01% out-of-system (90.83% vs. 86.82%). This gap suggests that our specification-based design is more robust to distribution shifts than end-to-end routers. By enriching query context with benchmark signals rather than relying on benchmark-wide supervision, our hierarchical structure (Section 4.3) generalizes better to unseen benchmark pools.

Comparison with a Single Large Model. Compared to the GPT-4.1 baseline, our method improves overall accuracy by 2.77% in-system and 4.54% out-of-system, despite routing only among lightweight agents. These results indicate that routing specialized agents can match or exceed a single general-purpose model by leveraging domain-specific strengths. While this advantage depends on the agent pool's composition, it highlights the efficiency of modular specialized systems over monolithic models for these benchmarks.

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, with results reported in Table 3.

Fusion benefits from combining complementary signals. Our scoring mechanism integrates similarity-based matching with confidence-aware signals (Section 3.2) to jointly capture both the relevance of an agent to the query and the reliability of its expected performance. Removing similarity fusion (*w/o* $\bar{s}_f$) reduces accuracy by 0.20 (in-system) and 0.78 (out-of-system), while removing confidence-aware fusion (*w/o* $\bar{g}_f$) leads to drops of 0.37 and 1.55 points. 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 and thus identifies agents with more dependable profiled behavior. Notably, the ablation gaps are consistently larger on out-of-system benchmarks, suggesting that confidence-aware aggregation and similarity fusion are important for improving robustness under out-of-system generalization.

Table 3: Ablation on in-/out-of-system benchmarks. Reported: Acc. (%) and absolute 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

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

Active profiling improves specification quality. Disabling active profiling and building specifications from randomly sampled instances under the same budget (*w/o active prof.*) reduces accuracy 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 out-of-system gap is consistent with the intuition that active profiling matters more when the evaluation distribution differs from the system-maintained benchmarks.

4.4 Analysis

We further analyze practical aspects of our approach, including computational cost (Appendix D.4), specification storage (Appendix D.5), and scalability with system size. Due to space constraints, the first two analyses are deferred to the appendix and focus on scalability below.

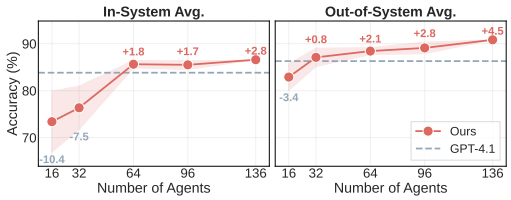

Figure 2: Scalability w.r.t. system size: in-/out-of-system average accuracy (shaded: std over 3 random subsets), with gains over GPT-4.1.

Scalability. Our plug-and-play design scales naturally with system size: adding agents only requires constructing specifications offline, without retraining any centralized selector. Figure 2 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. The advantage over GPT-4.1 further widens as the system size increases, confirming that our method maintains robust and scalable specification-based matching even in larger and more diverse agent pools.

5 Related Work

Learnware Specifications. The learnware paradigm (Zhou, 2016; Zhou & Tan, 2024) envisions an open *dock system* that hosts reusable models with compact *specifications*, enabling privacy-preserving identification and reuse without exposing raw data (Lei et al., 2024). Learnware specifications for conventional ML often summarize training data distributions via RKME (reduced kernel mean embedding) (Wu et al., 2023), and support identification via specification matching in a learnware system (Liu et al., 2024; Tan et al., 2024a;b). For language-model learnwares, recent work further explores PAVE (parameter vector) specifications (Tan et al., 2025; Shi et al., 2026) to represent model capabilities in a shared parameter space, conceptually related to *task vectors* that view task adaptation as directions in weight space (Ilharco et al., 2023; Ortiz-Jimenez et al., 2023). 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 (Shen et al., 2023), 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 (Jiang et al., 2023) combines outputs from multiple models, while FrugalGPT (Chen et al., 2023) 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 (Ong et al., 2025). Retrieval-style baselines retrieve the nearest samples and select the model with the best average per-

formance on those neighbors (Shnitzer et al., 2023; Hu et al., 2024). More recent proposals learn compact *model embeddings* from benchmark evaluation matrices and train lightweight predictors (EmbedLLM) (Zhuang et al., 2025), or derive *capability summaries* from benchmark performance and prompt a lightweight LLM to estimate per-query success probabilities (Model-SAT) (Zhang et al., 2025). 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.

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 (Lu et al., 2025; Li et al., 2025), and methods such as ReAct (Yao et al., 2022) and Reflexion (Shinn et al., 2023) improve tool use and iterative refinement. Tool-use capabilities are advanced by approaches such as Toolformer (Schick et al., 2023) and Search-R1 (Jin et al., 2025), which train LMs to decide when and how to invoke tools during inference. Meanwhile, multi-agent frameworks (Li et al., 2023; Hong et al., 2024) 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.

Acknowledgments

This work was supported by Jiangsu Science Foundation Leading-edge Technology Program (BK20232003) and Collaborative Innovation Center of Novel Software Technology and Industrialization. Zhi-Hao Tan was supported by JiangsuSF (BK20251215) and Jiangsu Funding Program for Excellent Postdoctoral Talent. 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 helpful comments.

REFERENCES

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *arXiv:1905.13319*, 2019.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

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

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. X-armed bandits. *Journal of Machine Learning Research*, 12(5):1655–1695, 2011.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv:2305.05176*, 2023.

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

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *Proceedings of the 9th International Conference on Learning Representations*, 2021a.

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

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

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. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proceedings of the 12th International Conference on Learning Representations*, 2024.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *Proceedings of the 10th International Conference on Learning Representations*, 2022.

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

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

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 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*, pp. 14165–14178, 2023.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv:2503.09516*, 2025.

Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William Cohen, and Xinghua Lu. 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*, pp. 2567–2577, 2019.

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

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

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems 36*, pp. 51991–52008, 2023.

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

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

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

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

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In *Proceedings of the 13th International Conference on Learning Representations*, 2025.

OpenAI. GPT-4 technical report. *arXiv:2303.08774*, 2023.

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

Ankit Pal, Logesh Kumar Umapathi, and Malaikannan Sankarasubbu. Medmcqa: A large-scale multi-subject multi-choice dataset for medical domain question answering. In *Proceedings of the Conference on Health, Inference, and Learning*, pp. 248–260, 2022.

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

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

Hao-Yu Shi, Zhi-Hao Tan, Zi-Chen Zhao, Yang Yu, and Zhi-Hua Zhou. A study on PAVE specification for learnware. In *Proceedings of the 14th International Conference on Learning Representations*, 2026.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24497–24524, 2025.

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

Tal Shnitzer, Anthony Ou, Mírian Silva, Kate Soule, Yuekai Sun, Justin Solomon, Neil Thompson, and Mikhail Yurochkin. Large language model routing with benchmark datasets. *arXiv:2309.15789*, 2023.

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

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. Beimingwu: A learnware dock system. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5773–5782, 2024b.

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

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

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

Qianqian Xie, Weiguang Han, Zhengyu Chen, Ruoyu Xiang, Xiao Zhang, Yueru He, Mengxi Xiao, Dong Li, Yongfu Dai, Duanyu Feng, et al. The finben: A holistic financial benchmark for large language models. *arXiv:2402.12659*, 2024.

Linyi Yang, Eoin Kenny, Tin Lok James Ng, Yi Yang, Barry Smyth, and Ruihai Dong. Generating plausible counterfactual explanations for deep transformers in financial text classification. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 6150–6160, 2020.

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

Yi-Kai Zhang, De-Chuan Zhan, and Han-Jia Ye. Capability instruction tuning. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence*, pp. 25958–25966, 2025.

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

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

Richard Zhuang, Tianhao Wu, Zhaojin Wen, Andrew Li, Jiantao Jiao, and Kannan Ramchandran. Embedllm: Learning compact representations of large language models. In *Proceedings of the 13th International Conference on Learning Representations*, 2025.

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.
	$D_m = \{(x_i, y_i)\}_{i=1}^{N_m}$	The m-th benchmark consisting of N_m query-answer pairs.
	$E : \mathcal{X} \to \mathbb{R}^d$	Embedding function mapping queries to d-dimensional vectors.
	$Q : \mathcal{Y} \times \mathcal{Y} \to [0, 1]$	Quality function measuring the quality of responses vs. reference answers.
Constructive Specification	$\mathcal{P}_f(D)$	Performance profile of f on benchmark instances with $Q(f(x), y) \geq \delta$.
	δ	Quality threshold determining successful handling of a given sample.
	$h(\cdot; \theta_0)$	Pre-trained base model as the base for computing parameter vectors.
	$\boldsymbol{\tau}_f(D)$	Parameter vector specification fitted on profile $\mathcal{P}_f(D)$ w.r.t. the base model.
	$\mathcal{T}_f(D)$	Hierarchical specifications: global and cluster-specific components.
	$\mathcal{S}_f(\mathcal{D})$	Agent-level constructive specification over the benchmark collection $\mathcal{D}$.
	$n_{\min}$	Minimum profile size required for 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 for agent invocations during capability profiling.
	$U_f(x')$	Optimistic estimate for unevaluated x' induced by evaluated set $\mathcal{V}$.
Agent Search	x^*	User query submitted to the 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 ALGORITHM DETAILS

This section provides the complete pseudocode for the three core procedures introduced in Section 3. Appendix B.1 presents the budget-efficient capability profiling algorithm, Appendix B.2 describes the hierarchical PAVE specification generation, and Appendix B.3 details the specification-based agent search at inference time.

B.1 BUDGET-EFFICIENT CAPABILITY PROFILING

Algorithm 1 discovers the performance profile $\mathcal{P}_f(D)$ of an agent f under a strict evaluation budget $B \ll N$. The benchmark is first preprocessed by splitting instances into easy/hard partitions via a baseline model and compressing the easy partition with K-Medoids clustering to remove redundancy. A set of k diverse seeds is then evaluated to bootstrap coarse coverage of the instance space. The main loop follows an *optimism-in-the-face-of-uncertainty* strategy (Auer et al., 2002; Kleinberg et al., 2008): each unevaluated candidate is assigned an optimistic upper bound $U^*(x')$ derived from a Lipschitz envelope over observed scores, and candidates are organized in a lazy max-heap that always expands the most promising one first. Candidates whose upper bounds fall below the quality threshold δ are safely pruned without evaluation (Theorem 3.3 and Corollary C.1). After each new

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 $\mathcal{Q} \leftarrow \emptyset,\ \mathcal{V} \leftarrow \emptyset,\ u_f(\cdot) \leftarrow \emptyset$;
Initialize $U^*(x) \leftarrow +\infty$ implicitly for unseen $x \in D'$;
$\mathcal{S}_{\text{init}} \leftarrow \text{K-Medoids}(\{E(x) \mid (x, y) \in D'\}, k)$;
$\text{Push}(\mathcal{Q}, (U^*(x), x))$ for all $x \in \mathcal{S}_{\text{init}}$;
while $|\mathcal{V}| < B$ ***and*** $\mathcal{Q} \neq \emptyset$ **do**
  $(U_{\text{key}}, x) \leftarrow \text{PopMax}(\mathcal{Q})$;
  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 $x' \in \text{k-NN}(x, D', n)$ **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 $U^*(x') \geq \delta$ **then**
        $\text{Push}(\mathcal{Q}, (U^*(x'), x'))$;
      end
    end
  end
end
return $\mathcal{P}_f(D) = \{(x, y) \in D \mid x \in \mathcal{V},\ u_f(x) \geq \delta\}$

evaluation, the upper bounds of nearby candidates are tightened, progressively concentrating the remaining budget on regions that are simultaneously high-performing and under-explored.

B.2 Hierarchical PAVE Specification Generation

Given the profiled set $\mathcal{P}_f(D)$, Algorithm 2 compiles it into a hierarchical specification $\mathcal{T}_f(D)$ for cross-agent similarity matching. Each profiled instance is first assigned a discriminative weight, where higher weights are allocated to baseline-hard cases to better separate agent capabilities. A global PAVE vector $\boldsymbol{\tau}_f(D)$ is then computed by fitting a parameter update to the shared base model $h(\cdot; \theta_0)$ on the weighted profile (Eq. equation 3). Because all agents share the same θ_0, the resulting vectors are natively comparable via cosine similarity. To preserve within-domain heterogeneity, the profile is further partitioned into C semantic clusters via K-Means on sample embeddings, and a cluster-level PAVE vector $\boldsymbol{\tau}_f(D, c)$ is extracted for each sufficiently large cluster. The final specification $\mathcal{T}_f(D) = \{\boldsymbol{\tau}_f(D)\} \cup \{\boldsymbol{\tau}_f(D, c)\}_{c \in \mathcal{C}_f}$ (Eq. equation 4) provides multi-granularity representations that enable robust query-level matching even when the user-side signal is limited.

B.3 Specification-Based Agent Search

At inference time, Algorithm 3 selects the best agent for a user query x^* without any additional agent evaluations. The query is first anchored to the most relevant benchmark D^* by averaging embedding similarities with nearest neighbors in each benchmark (Eq. equation 5). A query specification $\boldsymbol{\tau}_{x^*}$ is then constructed by applying PAVE on the retrieved neighborhood $\mathcal{R}(x^*) \subseteq D^*$ with relevance-reweighted difficulty weights (Eq. equation 6), enriching the sparse query with task context from the benchmark pool. For each candidate agent f, the query specification is matched against its hierarchical vector set $\mathcal{T}_f(D^*)$: the top-k cosine similarities are aggregated with geometrically decaying weights (Eq. equation 7), so that the best-matching granularity dominates while secondary matches still contribute. Finally, the similarity score is fused with the agent's baseline-hard competence via a confidence-adaptive weight α (Eq. equation 8)—when benchmark coverage is strong α favors simi-

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. equation 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
$\boldsymbol{\tau}_f(D) \leftarrow \text{PAVE}(\mathcal{P}_f(D), \text{NORMALIZE}(\{w_i\}), \theta_0)$ by Eq. equation 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** $\mathcal{T}_f(D)$;
$\{c_i\} \leftarrow \text{K-MEANS}(\{E(x_i)\}, C)$;
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. equation 3 on $\mathcal{P}_f(D, c)$;
  $\mathcal{T}_f(D) \leftarrow \mathcal{T}_f(D) \cup \{\boldsymbol{\tau}_f(D, c)\}$;
end
return $\mathcal{T}_f(D)$

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. equation 5 and get the local neighborhood $\mathcal{R}(x^*)$;
Construct the query specification $\boldsymbol{\tau}_{x^*}$ by Eq. equation 6;
foreach $f \in \mathcal{F}$ **do**
  if $\mathcal{T}_f(D^*) = \emptyset$ **then continue**;
  Compute s_f by Eq. equation 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. equation 8 for each agent f;
return $\{(f, \text{score}_f)\}$ *sorted by* score_f *descending*;

larity, and when coverage is weak it gracefully falls back to the competence prior—yielding a robust ranking across diverse query conditions.

C Theoretical Analysis of Active Profiling

This section provides complete proofs for the theoretical results stated in Section 3.1. The analysis follows a three-part logical chain: we first establish that the optimistic envelope U_f is a valid upper bound on true performance (Theorem 3.3), then show that threshold-based pruning safely excludes unpromising candidates without false negatives (Corollary C.1), and finally bound the computational overhead of lazy priority-queue maintenance (Proposition C.2).

C.1 Optimistic Upper-Bound Guarantee

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: $u_f(x') \le u_f(x) + L \cdot d(x', x)$. Since $\beta \ge L$, we have

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

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

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

which completes the proof. □

C.2 Safe Pruning via Threshold Test

Corollary C.1 (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 C.1 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.

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

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

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

C.3 Budget-Sensitive Computational Overhead

Proposition C.2 (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,*

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

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

Proposition C.2 complements Corollary C.1 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.

Proof. Each evaluation adds one new element into $\mathcal{V}$, so $|\mathcal{V}| \le B$. 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,

$$\#\mathrm{Push} \le k + n|\mathcal{V}| \le k + nB.$$

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

$$\#\mathrm{PopMax} \le \#\mathrm{Push} \le k + nB.$$

With a binary heap, the total time complexity is

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

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

D Additional Experimental Details and Results

This section provides comprehensive details on our experimental setup and presents supplementary results that further support the findings from Section 4.

D.1 Agent System Details

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 (Li et al., 2023) 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, deephubwiki, 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 (Shi et al., 2025) 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.

D.2 Benchmark Details

D.2.1 In-System Benchmarks

We use six benchmarks across three domains (mathematics, healthcare, and finance) to build the agent system. These benchmarks serve dual purposes: their training splits are used for agent profiling and specification construction, while their test splits are used for evaluating in-system performance. The selected benchmarks cover diverse reasoning patterns and task formats, including multi-step mathematical reasoning, domain-specific knowledge retrieval, and financial sentiment classification, allowing us to evaluate methods across heterogeneous task settings.

- *GSM8K* (Cobbe et al., 2021) is a benchmark for evaluating grade-school mathematical reasoning. It contains 8.5K high-quality math word problems requiring multi-step numerical reasoning and arithmetic operations. The dataset consists of 7,473 training samples and 1,319 testing samples.
- *MATH* (Hendrycks et al., 2021b) comprises 12,500 challenging problems from high school mathematics competitions, covering algebra, geometry, number theory, and calculus, and is designed to assess advanced mathematical reasoning skills. It contains 7,500 training samples and 5,000 testing samples.
- *MathQA* (Amini et al., 2019) is a large-scale dataset for mathematical reasoning and quantitative problem solving. It consists of natural language math word problems paired with structured equation representations and final answers, covering arithmetic, algebra, and basic quantitative reasoning tasks. It contains 29,837 training samples and 2,985 testing samples.

- *MedMCQA* (Pal et al., 2022) is a large-scale medical QA dataset with multiple choices derived from medical entrance examinations (AIIMS/NEET), covering 21 medical subjects and over 2.4k healthcare topics. Each question has 4 answer choices and is accompanied by an explanation. It evaluates a model's general medical knowledge and reasoning capabilities. It contains 182,822 training samples and 6,150 testing samples.
- *PubMedQA* (Jin et al., 2019) is a closed-domain biomedical QA dataset in which each question is paired with a relevant PubMed abstract, requiring yes/no/maybe answers based on scientific evidence. It assesses a model's ability to comprehend and reason over scientific biomedical literature. It contains 450 training samples and 500 testing samples.
- *FPB* (Malo et al., 2014) is a financial sentiment analysis dataset consisting of sentences extracted from financial news and reports, annotated with positive, negative, or neutral sentiment labels. This dataset contains 3,100 training samples and 970 testing samples.

D.2.2 Out-of-System Benchmarks

To evaluate generalization capability, we select benchmarks from the same three domains (mathematics, medicine, and finance) but with different data distributions and task characteristics than the in-system benchmarks. These benchmarks are used exclusively for testing and are never seen during agent profiling or specification construction.

Mathematics Domain. We use four mathematics-related subsets from MMLU (Hendrycks et al., 2021a), a large-scale multitask benchmark composed of multiple-choice questions:

- *Abstract Algebra* (100 samples): Tests fundamental concepts in abstract algebra including 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 healthcare-related subsets from MMLU (Hendrycks et al., 2021a) that cover various aspects of medical knowledge:

- *Anatomy* (135 samples): Tests knowledge of human anatomical structures and systems.
- *Clinical Knowledge* (265 samples): Evaluates clinical reasoning and medical decision-making.
- *College Biology* (144 samples): Covers biological concepts at the undergraduate level.
- *College Medicine* (173 samples): Assesses medical knowledge taught in medical school curricula.
- *Medical Genetics* (100 samples): Tests understanding of genetic principles 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 (Xie et al., 2024) to evaluate performance across different forms of financial understanding and reasoning:

- *MA* (Yang et al., 2020) (500 samples): Sentences collected from Mergers and Acquisitions (M&A) news articles or social media posts such as tweets. The task is to classify whether the mentioned deal was completed or remained a rumor.
- *German Credit* (Hofmann, 1994) (200 samples): A classic credit scoring dataset where the task is to classify individuals as "good" or "bad" credit risks based on historical customer attributes.

D.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.

D.3 Evaluation Protocol and Metrics

We report *agent identification accuracy*: the higher the accuracy, the better the method is at selecting an agent that actually 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 implemented in a task-appropriate way for each benchmark. For mathematical reasoning (GSM8K, MATH, MathQA and MMLU math subsets), we extract the final numerical or symbolic answer from the model output (e.g., the last number or expression before a newline or period) and set $Q = 1$ when it matches the ground-truth under normalized exact match (numerical tolerance and string normalization following common practice), and $Q = 0$ otherwise. For multiple-choice benchmarks (MedMCQA, PubMedQA, MMLU subsets, MA, German Credit), we parse the model output for the selected option (e.g., A/B/C/D or the option text) and set $Q = 1$ when it matches the gold label and $Q = 0$ otherwise. For sentiment classification (FPB), we map the model output to one of the sentiment labels and set $Q = 1$ when it matches the reference and $Q = 0$ otherwise. This protocol is consistent across all contenders and our method. Each method only differs in how the agent is selected for a given query. 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 of agents.

D.4 Computational Overhead

Offline cost. 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: if the agent performs poorly on the benchmark, profiling terminates early once the success set is deemed sufficient or the agent is unlikely to improve, yielding cost-efficient profiling. 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 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.

Online cost. At serving time, given a user 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 current prototype, step (1) takes under 3 seconds and step (2) takes under 0.1 seconds. The primary contribution of this work is *identification accuracy* rather than latency minimization. We therefore do not optimize the online pipeline for speed in this study. Importantly, end-to-end routing latency in practice is dominated by the selected agent's response time (often tens of seconds for reasoning or long-generation tasks), so our selection overhead remains a small fraction of the user-visible delay, a design choice that prioritizes correctness of routing over marginal gains in selection latency.

The current numbers leave substantial room for acceleration. *(1) benchmark anchoring and retrieval of nearest neighbors:* lighter retrieval models, approximate nearest neighbor search, and parallelization can reduce cost without changing the method. *(2) specification generation:* we already adopt low-rank PAVE (Shi et al., 2026). Further gains can come from a lighter quantized base model for extraction, more efficient PEFT (e.g., structured adapters, pruning), fewer retrieved nearest neighbors, and caching of user specifications for similar queries. *(3) specification matching:* parallel computation of similarity scores or an efficient vector database can further accelerate matching without affecting the performance. Together, these directions show that the framework is compatible with standard systems optimizations while the present study focuses on establishing that the *identification* mechanism itself is accurate and scalable.

D.5 Specification Size and Storage

In this experiment, each agent's specification is a set of hierarchical PAVE vectors, where each vector encodes a capability profile as a parameter update to the shared base model and is stored as a LoRA-

style low-rank factorization for space efficiency. For a given benchmark, the specification comprises one global vector summarizing overall competence and multiple cluster-level vectors capturing fine-grained capability sub-patterns, all residing in a unified parameter space that enables direct cross-agent comparison via cosine similarity. 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 3 (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). Figure 3 (right) confirms near-linear scaling with benchmark count ($y = 15.74x + 9.28$, $R^2 = 0.935$), making the storage growth predictable as the system-maintained benchmark collection expands.

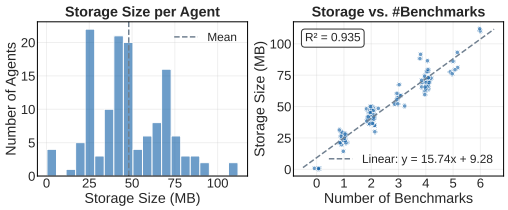

Figure 3: Specification size and storage (left: distribution; right: vs. benchmark count).

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.

D.6 Contenders Implementation Details

We provide implementation details for each contender method:

- *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* (Shnitzer et al., 2023; Hu et al., 2024): We retrieve the $k = 16$ most similar training queries and select the agent with the best average performance.
- *EmbedLLM* (Zhuang et al., 2025): We learn 232-dimensional agent embeddings from the query–agent performance matrix.
- *Model-SAT* (Zhang et al., 2025): 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.

D.7 Our Implementation Details

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