BenchRouter: Routing, Not Rewriting, for Heterogeneous LLM Agent Evaluation

Anonymous ACL submission

Abstract

LLM-agent evaluation is increasingly bottlenecked by systems integration rather than benchmark availability. Modern agent benchmarks ship with their own services, browser state, container workflows, and result schemas, while unified frameworks often integrate them by rewriting evaluator logic into platform-specific tasks. For environment-heavy benchmarks, this adaptation introduces three recurring costs: framework lock-in, rewriting risk, and upstream synchronization burden.

We present BenchRouter, an evaluation router for heterogeneous LLM agent benchmarks. BenchRouter treats each benchmark as an opaque workload and standardizes only the execution boundary through manifests, environment variables, and `result.json`. Its architecture combines a minimal benchmark contract, an execution router with Simple and DinD modes, on-demand environment building, and Run/Job aggregation over task-level results.

We evaluate BenchRouter on eight benchmarks chosen to cover integration regimes in agent evaluation. Across these integrations, the router-facing layer remains small (median 210 lines of bridge code per benchmark) while upstream evaluator logic stays outside the platform and maintenance remains concentrated at the execution boundary rather than in a platform-owned port. For current environment-heavy benchmarks, routing is a practical alternative to rewriting evaluators into a shared framework. We use coverage, fidelity, and adaptation surface as a lens for comparing evaluation systems.

1 Introduction

LLM-agent benchmarks are proliferating faster than the infrastructure needed to run them. Recent surveys catalog more than one hundred benchmarks for web navigation, software engineering, tool use, multimodal reasoning, and scientific or conversational tasks. But benchmark availability does not imply benchmark usability. In practice, even running a small set of agent benchmarks often means maintaining separate launchers, environments, result schemas, and operational playbooks.

This mismatch between benchmark availability and benchmark adoption is increasingly a systems problem. Many modern agent benchmarks are not just datasets plus prompts; they are complete runtime stacks. MCPMark (Wu et al., 2025) couples evaluation to live tool servers and backing services such as PostgreSQL. SWE-bench Pro (Deng et al., 2025) couples evaluation to per-instance software environments and patch execution. WebArena (Zhou et al., 2024) couples evaluation to browser automation over stateful web applications. In other words, the hard part is often not calling the model. The hard part is preserving the benchmark's execution boundary, service graph, and scoring logic.

This shift is also recent enough to matter as a contemporary evaluation-systems problem. Much earlier LLM evaluation was still largely prompt-bounded: shared model wrappers, shared datasets, and shared scorers were often enough. The newer wave of agent benchmarks changes the locus of complexity. Integration work is moving away from prompt formatting and toward execution coordination across services, browser state, and benchmark-owned container workflows.

Existing unified evaluation frameworks improve ergonomics at the top level by offering one CLI, one configuration system, or one task abstraction. Their trade-off is that new benchmarks are often adapted into framework-specific plugins or evaluator classes. That adaptation cost is modest for prompt-oriented tasks, but much higher for benchmarks whose evaluator is entangled with container setup, multi-process orchestration, or benchmark-local service dependencies. This creates a tension between *unification* and *fidelity*: the more a framework asks benchmarks to conform to a shared ab-

straction, the easier it becomes to run them together, but the harder it becomes to preserve the benchmark's original execution path.

BenchRouter studies a different design point: benchmark integration as *routing* rather than *rewriting*. Benchmarks are registered as opaque workloads, typically packaged in their own containers. The platform standardizes only the execution boundary: manifests describe tasks, environment variables carry model configuration, and each benchmark exports a standardized `result.json`. The platform therefore owns orchestration rather than benchmark evaluation logic.

The main claim of BenchRouter is not that it eliminates integration effort, but that it relocates that effort to a smaller and more stable boundary. In the public artifact, adding a benchmark typically requires manifests plus a thin bridge script, while evaluator logic remains in upstream code.

Our evaluation target is therefore a systems target rather than a benchmark-score target. Three hypotheses guide the evaluation. **H1 (coverage)**: the contract covers the main integration regimes we observed in our sample of eight environment-heavy benchmarks, rather than validating coverage of the ecosystem as a whole. **H2 (fidelity, suggestive)**: routing preserves benchmark behavior within a small margin of the benchmark-owned evaluation path; in the MCPMark fidelity check, the absolute gap between routed and official scores stays under 1.2 points (mean 0.82) across all five server families, with residuals attributable to run-to-run sampling variance rather than to evaluator divergence. **H3 (footprint)**: under routing, adaptation cost remains concentrated at the boundary, which we operationalize through bridge LOC, environment-preparation logic, and operational metrics rather than platform-owned scorer rewrites.

Contributions. This paper makes three contributions.

- We formulate multi-benchmark LLM-agent evaluation as a systems integration problem and identify three recurring forms of *adaptation cost*: framework lock-in, evaluator-rewriting risk, and upstream synchronization burden, then study their consequences through coverage, fidelity, and adaptation-surface evidence (§3–§5).
- We present the design of BenchRouter, including its execution architecture, benchmark contract, execution router, environment manager, Run/Job model, and two execution modes organized around the relation between the evaluation container and the acting environment (§4).
- We report an integration study across eight heterogeneous benchmarks chosen to cover representative integration regimes, show that the router-facing layer has median 210 lines of bridge code with 0 upstream scorer modifications, and organize the evidence around coverage (the contract covers all eight benchmarks we studied, as an observation on a high-friction sample rather than a validation on the ecosystem), fidelity ($|\Delta_{BR}| < 1.2$ points across all five MCPMark server families, attributable to run-to-run sampling variance under an unchanged upstream evaluator), and adaptation surface (§5).

The resulting system is open source. Across the eight integrations we study, the benchmark-specific layer is typically a single short bridge script (median 210 lines of Python; Table 3). For benchmark authors, this changes the integration request from "rewrite your evaluator into our framework" to "expose a container entry point and a result file." For benchmark users, it offers one CLI surface across otherwise heterogeneous benchmark families.

2 Related Work

We situate BenchRouter among unified evaluation frameworks for prompt-oriented tasks, benchmark suites that package multiple agent environments behind one harness, and general evaluation platforms with their own task DSL or plugin interface. BenchRouter is not a new benchmark suite and does not ask benchmarks to be rewritten into a platform-owned evaluator abstraction.

Static evaluation frameworks. HELM (Liang et al., 2022), *lm-evaluation-harness* (Gao et al., 2024), and BIG-bench (Kazemi et al., 2025) established the dominant pattern for prompt-oriented model evaluation: wrap the model behind a common inference interface and score it over a large task library. These systems solved an important comparability problem, but their unit of abstraction remains close to "task as prompt set." That abstraction becomes strained when the benchmark owns its own services, browser state, or containerized control flow.

Benchmark suites and platforms. Another line of work packages multiple environments behind one benchmark suite or harness. AgentBench (Liu et al., 2024), AgentBoard (Ma et al., 2024), and MLAgentBench (Huang et al., 2023) all improve comparability by placing diverse agent tasks behind a common launcher, and OSWorld (Xie et al., 2024) standardizes around a desktop-VM execution model for GUI agents. These projects are valuable as benchmark suites, but they generally assume ownership of the benchmark harness itself. BenchRouter instead targets independently developed benchmarks. Inspect (UK AI Security Institute et al., 2024) is the closest general evaluation platform, but its integrations port benchmark logic into a platform-owned Python task definition and runtime. BenchRouter instead keeps benchmark logic upstream and standardizes only launch-time configuration and result collection. OpenCompass (Contributors, 2023) and VLMEvalKit (Duan et al., 2024) are strongest for prompt-based tasks rather than benchmarks that need their own Docker daemon or multi-service runtime. SWE-bench's official harness (Jimenez et al., 2024) already runs each task in a per-instance Docker image, foreshadowing BenchRouter's DinD mode at the level of a single benchmark; BenchRouter generalizes this pattern to a multi-benchmark router.

Workflow and orchestration systems. BenchRouter also draws on ideas familiar from workflow orchestration systems such as Airflow, Snakemake, Nextflow, Kubernetes Jobs, or Argo Workflows: explicit task boundaries, containerized execution, retryable units of work, and run-level aggregation. The difference is that those systems are domain-agnostic schedulers, whereas BenchRouter specializes the contract for benchmark execution. It owns model-endpoint injection, benchmark manifests, result standardization, and the distinction between evaluation containers and acting environments. In this sense, BenchRouter is not a generic workflow engine with a new CLI, but an evaluation-oriented routing layer built on top of orchestration principles.

Standards for reusable evaluation environments. Recent proposals push standardization further upward by asking benchmarks to adopt richer typed APIs once and then expose them everywhere. CUBE (Lacoste et al., 2026) factorizes evaluation into task, benchmark, package, and registry abstractions, targeting a "wrap once, use everywhere" workflow with stronger shared semantics. vla-eval (Choi et al., 2026) similarly decouples model inference from benchmark execution through a narrow protocol, but does so in the more constrained VLA domain and pairs that protocol with stronger throughput and reproducibility analysis. BenchRouter takes the opposite trade-off: it minimizes author-side adoption cost by standardizing only the execution boundary, at the cost of fewer shared evaluation primitives. Adjacent work reinforces the same operational lesson seen in our integrations: SWE-World (Sun et al., 2026) explores Docker-free surrogates as an alternative to real-execution pipelines, and Multi-Docker-Eval (Fu et al., 2025) benchmarks the ability of agents to construct runnable Docker environments—both suggest that orchestration, cache locality, and environment construction often dominate practical adaptation cost.

3 Problem Formulation

The main difficulty in multi-benchmark agent evaluation is not lack of benchmarks, but the work required to make heterogeneous benchmarks runnable under one operational surface. We refer to this systems burden as *adaptation cost*. In the frameworks we surveyed, that cost appears in three recurring forms that often compound in practice.

(I) Framework lock-in. The first cost is simple: before you can run the benchmark, you must first learn and satisfy the platform's own interface. Most multi-benchmark platforms expose a framework-specific registration interface: a base class to inherit, a registry to populate, or a task schema to conform to. For prompt-oriented benchmarks this may be reasonable ergonomically, but for benchmark authors who already have a working evaluation entry point, the port introduces a new dependency surface without adding new evaluation signal. The benchmark is now coupled not only to its own codebase, but also to the framework that hosts it.

(II) Rewriting risk for tightly-coupled pipelines. The second cost is risk introduced by rewriting a benchmark that already works. Many agent benchmarks intertwine model interaction with benchmark state, service orchestration, or benchmark-owned control flow. Examples include live database interaction, browser automation, and patch-and-test loops inside benchmark-specific containers. For such benchmarks, adapting eval-

Framework	Lock-in	Rewriting risk	Sync burden	Heavy runtime support	Reusable scorer library
BenchRouter	Low	Low	Low–Medium	Yes	None
One-Eval*	N/A	N/A	N/A	N/A	N/A
Inspect	High	High	High	Limited	Yes (task DSL)
lm-evaluation-harness	High	High	High	No	Yes (shared scorers)
OpenCompass	High	High	High	No	Yes (config + scorers)
VLMEvalKit	High	High	High	No	Yes (multimodal scorers)

Table 1: Positioning of BenchRouter relative to representative evaluation systems along the three adaptation costs of §3. Lock-in measures whether the benchmark must import or inherit framework-specific abstractions; rewriting risk measures whether the evaluator path is re-expressed in platform code; sync burden measures how much of that platform code must change when the upstream benchmark evolves. Heavy runtime support indicates whether environment-heavy or nested-Docker benchmarks are first-class. Reusable scorer library captures a feature that rewriting-oriented frameworks provide and BenchRouter does not: a built-in, cross-benchmark library of scorers, agents, and task templates that downstream benchmarks can import instead of supplying their own. *One-Eval (Shen et al., 2026) is a planning and reporting layer that decides *what* to evaluate; it operates above any routing/rewriting choice and is complementary to BenchRouter rather than an alternative (see §3).

uation into a single framework-level `evaluate()` abstraction is not just syntactic wrapping; it often requires re-expressing orchestration logic. That rewrite raises the risk of *scoring divergence*: the adapted pipeline is no longer behaviorally equivalent to the benchmark's intended execution path, so cross-paper comparison becomes harder to trust.

(III) Upstream synchronization burden. The third cost appears later: once the benchmark changes upstream, the adaptation layer must change with it. Benchmarks evolve after publication: task splits change, scorer bugs are fixed, dependencies move, and new variants appear. If a benchmark has been deeply adapted into a framework, each upstream change must be mirrored through the adaptation layer to preserve comparability with community baselines. The maintenance burden therefore grows not only with the number of integrated benchmarks, but also with the amount of benchmark logic that has been copied or restructured. A thinner router boundary does not eliminate maintenance, but it reduces how much benchmark semantics the platform itself must own.

BenchRouter addresses these costs with one design principle: preserve the benchmark's execution boundary and standardize only the routing boundary. The platform does not ask benchmark code to import a framework library or inherit a platform-specific evaluator. Instead, the benchmark remains an opaque workload, while the orchestration layer standardizes model configuration, execution, and result collection. Table 3 shows that, in our integrations, the benchmark-specific layer is typically narrow even for environment-heavy benchmarks.

4 System

BenchRouter is designed around one thesis: many current agent benchmarks can share one orchestration layer without sharing one evaluator abstraction. Rather than asking benchmarks to import platform libraries or translate their control flow into a shared `evaluate()` interface, BenchRouter standardizes only the execution boundary and treats each benchmark as an opaque workload.

At a high level, the system has four components: a benchmark contract, an execution router, an environment manager, and a Run/Job aggregator. The contract says what a benchmark must expose; the remaining components say how the platform executes that benchmark without taking ownership of its evaluator.

4.1 Benchmark contract

A BenchRouter benchmark is defined by a small public contract: a benchmark manifest, one task manifest per task, and an executable entry point that reads standardized inputs and emits standardized output. The goal is not to standardize benchmark internals, but to standardize enough of the boundary for scheduling, monitoring, and comparison.

Benchmark and task manifests. *benchmark.yaml* lists tasks at the benchmark root, while *task.yaml* in each task directory specifies the command, timeout, execution mode, and optional environment configuration. The *result_mapping* field is the only point of schema negotiation: it

Variable	Meaning
BENCHROUTER_MODEL_ENDPOINT	OpenAI-compatible base URL
BENCHROUTER_MODEL_NAME	Model identifier
OPENAI_API_KEY	API credential
BENCHROUTER_OUTPUT_DIR	Output path for result.json

Table 2: The benchmark-facing contract: four environment variables in, one JSON file out. Additional scheduler variables are infrastructural rather than part of the public protocol.

maps an existing benchmark-specific score field (*pass@1*, *accuracy*, etc.) to the standardized *overall* key without forcing a new internal result schema. The mapping does not recompute benchmark-local composite metrics. Instead, the benchmark remains responsible for deciding which exported field is authoritative as *overall*, while any additional metrics are preserved under *raw_metrics*. In other words, BenchRouter standardizes one top-level score for routing and aggregation, but it does not adjudicate benchmark-owned multi-metric semantics.

The four-variable / one-file contract. At runtime, the scheduler injects four environment variables into every container, and expects one JSON file in return:

At the low end, a valid task can be a shell script that calls a model endpoint and writes `{"overall": 0.85}`. At the high end, the same contract can wrap benchmarks that launch browser stacks or nested Docker workflows. There are no base classes to inherit, no plugin registration hooks, and no requirement that benchmark code import platform code.

A minimal contract, written precisely. We summarize the public interface as

$$\mathcal{C}_{BR} = (I,\ O,\ M,\ T),$$

where I is the injected model/runtime input (the four environment variables in Table 2); O is the `result.json` written under the output directory; M is the optional *result_mapping* from a benchmark-local score field to the standardized *overall* field; and T is the task manifest that specifies command, timeout, and execution mode. A benchmark is cheap to route when model traffic can be redirected through I, the benchmark can materialize a JSON-compatible result under O, and any benchmark-local acting environment can be classified as co-resident, remote external, or heavyweight hosted. This is the contract-level sense in which BenchRouter argues for routing rather than rewriting.

Scope of the current contract. The current contract is intentionally scoped to endpoint-style inference. It assumes that model configuration can be supplied as a base URL, model identifier, and credential, and that benchmark execution can return one benchmark-owned *overall* metric plus auxiliary raw fields. Richer interfaces such as tool/-function calling, streaming outputs, multimodal request schemas, or non-OpenAI transport protocols would likely require typed optional fields or a pluggable transport layer rather than only environment-variable injection.

Container as the only runtime contract. Containerization is the only runtime assumption the platform makes about benchmark internals. There is no global Python-version constraint, no requirement that all benchmarks share the same dependency stack, and no need for a benchmark to expose a framework-native API. This keeps incompatibilities local to the benchmark image rather than global to the evaluation campaign.

4.2 Execution architecture

BenchRouter separates user interaction, orchestration, and benchmark execution. The CLI and server expose a unified control plane, while the scheduler, registry, and container runtime preserve benchmark-local execution (Figure 1).

4.3 Execution router and Run/Job model

A benchmark execution produces one *Run* and one *Job* per task. Jobs are scheduled in parallel under a semaphore-bound concurrency limit. Run status is derived from child-job outcomes rather than from the first failure: COMPLETED (all jobs ok), PARTIAL (some ok, some failed), or FAILED (none ok). PARTIAL is deliberate: environment-heavy benchmarks fail for operational reasons as well as modeling reasons, so the router preserves completed work instead of collapsing a run to a single Boolean failure.

4.4 Execution modes

Each task selects one of two modes via a single field in *task.yaml*, distinguished by the relation between the *evaluation container* (evaluator, depen-

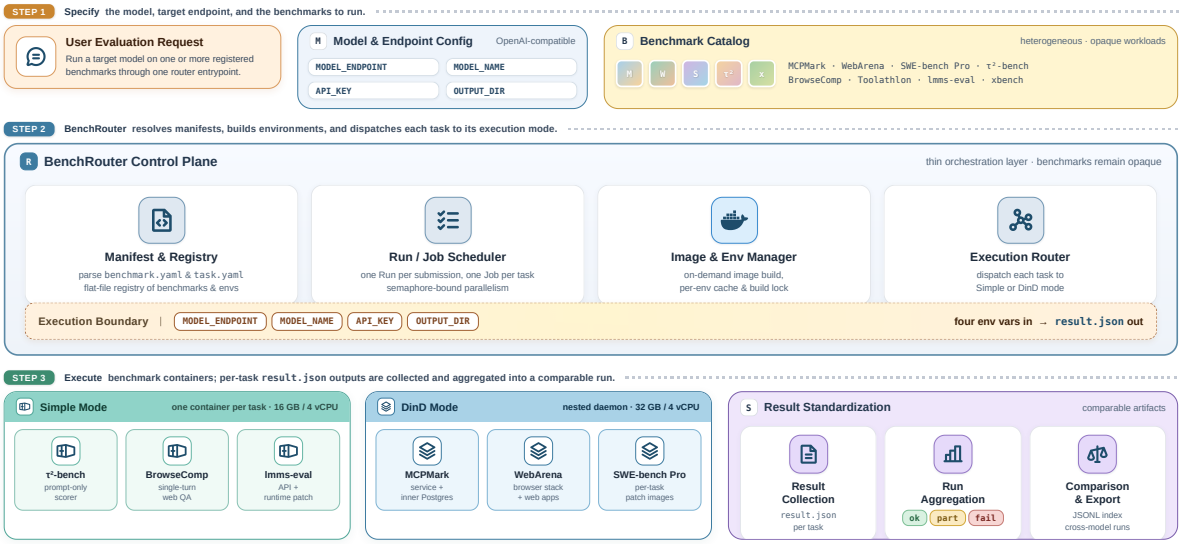

Figure 1: BenchRouter end-to-end flow. Three input streams (user request, model configuration, benchmark catalog) feed into the BenchRouter control plane, which standardizes a four-variable execution boundary and dispatches each task to either Simple or DinD mode; per-task `result.json` outputs are aggregated into completed, partial, or failed runs.

dencies, model calls, scoring) and the *acting environment* (filesystem, browser state, service stack). The modes cover whether that environment stays inside the evaluation container, lives remotely, or needs a heavier containerized stack.

Simple mode (default). One container, one task. The benchmark directory is mounted read-only at `/app/benchmark`; results are written to `/app/results`. Defaults are 16 GB RAM, 4 CPU cores, and host networking. This covers benchmarks that need only the evaluation container, those whose acting environment is light enough to co-locate with it, and those whose acting environment lives remotely.

Docker-in-Docker (DinD) mode. DinD is used when the acting environment is too heavy to co-locate with the evaluation container. Typical examples are browser-backed sites, multi-service tool environments, or benchmark-owned nested container workflows. Setting `docker: true` routes the task to DinD mode. Each task gets an isolated inner Docker daemon, via the Sysbox runtime where available and a privileged-container fallback otherwise. Pre-cached images listed in *docker_images* are loaded into the inner daemon to avoid repeated multi-gigabyte pulls. Defaults are 32 GB RAM and 4 CPU cores.

4.5 Environment, aggregation, and SDK runner

A benchmark directory may include a *Dockerfile* under `environments/⟨env⟩/`; on first use, the scheduler acquires a per-environment lock, builds, and caches. Concurrent jobs that depend on the same image wait on the build result rather than rebuilding. After all jobs finish, the scheduler reads each *result.json*, applies any per-task *result_mapping*, and appends a JSONL index. Inside every container, a small SDK runner (~60 lines) executes the user command, applies the field remap, and writes *result.json* with *benchmark_name*, *model_name*, *overall*, and *raw_metrics*; raw metrics are preserved verbatim.

5 Evaluation

We evaluate BenchRouter as an evaluation system rather than as a benchmark-results paper, against three hypotheses. **H1 (coverage)**: the contract covers the main integration regimes observed across our eight-benchmark sample, treated as a high-friction sample rather than an ecosystem coverage proof. **H2 (fidelity, suggestive)**: at the operating points in Table 4, routed scores remain within a small absolute gap from the benchmark-owned evaluation path across all five MCPMark server families. **H3 (footprint)**: the platform-specific adaptation surface stays narrow because benchmark semantics remain upstream. We address H1 with

the runtime taxonomy below, H2 with Table 4, and H3 with Table 3.

The benchmarks span MCP tool use, browsing, software engineering, multimodal QA, customer service, and contamination-resistant knowledge tasks. The acting environment falls into five relations: evaluator-only (lmms-eval (Zhang et al., 2024), τ^2-bench (Barres et al., 2025), BrowseComp (Wei et al., 2025), xbench (Chen et al., 2025)), co-resident lightweight (MCPMark Filesystem), remote external (MCPMark Notion), hosted heavy (Toolathlon (Li et al., 2025), WebArena), and benchmark-owned execution (SWE-bench Pro); Appendix Table 5 lists per-benchmark assignments. All eight satisfy contract conditions P1–P3; boundary cases (Appendix A) strain P3 rather than P1 or P2.

Beyond environment class, bridges also vary by integration form: Pattern A (5/8) wraps an upstream CLI; Pattern B (3/8) keeps the evaluator in process and patches only the model-call boundary (Appendix A.5). Case studies below proceed from lightest to heaviest.

Table 3 shows the same router boundary spans relations that range from no environment at all to heavyweight hosted stacks. The τ^2-bench row is heavier than other evaluator-only benchmarks because it preserves a separate user-simulator path and the benchmark's native metric interface rather than a single QA-style score call. Across all benchmarks, upstream scorer code is never modified, and what remains as platform-specific code is boundary code (bridge scripts, manifests, environment preparation) rather than evaluator semantics.

Contrast with rewrite-centric frameworks. The same contrast recurs across our three Inspect comparisons. At the evaluator-only end (§5.lmms-eval), Inspect's MMMU port re-expresses dataset, solver, and scorer ownership inside a platform task definition (286 LOC vs. BenchRouter's 192). At the benchmark-owned end (§5.3), its SWE-bench port re-owns image construction, patch application, and the test workflow (754 LOC vs. BenchRouter's 227); on MCPMark, Table 4 shows that the routed path preserves the official scores closely, consistent with leaving the upstream evaluator authoritative. We use Inspect because it publicly exposes the evaluator ownership boundary in code; prompt-bounded frameworks make the same trade-off without environment-heavy comparators.

5.1 lmms-eval

lmms-eval runs in *Simple* mode and already exposes a usable Python evaluation API, so the router only adapts the model-client boundary; the upstream evaluator targets a BenchRouter-managed endpoint without reproducing benchmark control flow.

A concrete code-level diff with Inspect. Inspect's public MMMU port reconstructs dataset loading, solver selection, and scorer selection inside a platform task definition:

```
@task
def mmmu_multiple_choice(...):
    combined_dataset = load_and_concatenate_datasets(...)
    return Task(
        dataset=multiple_choice_questions_inspect,
        solver=[prompt_template(...), multiple_choice()],
        scorer=choice(),
    )
```

In the BenchRouter bridge, the upstream lmms-eval evaluator remains intact and the router only redirects model transport before calling the benchmark's own evaluation entry point:

```
model_endpoint = os.environ["BENCHROUTER_MODEL_ENDPOINT"]
os.environ["OPENAI_API_BASE"] = model_endpoint
results = evaluator.simple_evaluate(
    model="openai",
    model_args=model_args,
    tasks=[task_name],
)
```

Inspect therefore owns more *evaluator semantics* (which dataset to load, which solver to run, which scorer to apply), while BenchRouter owns more of the *execution boundary* (container launch, endpoint injection, result collection) and leaves the benchmark's own evaluator authoritative.

5.2 MCPMark

MCPMark illustrates why environment-bearing agent benchmarks require more than a single execution mode. Its five public server families exercise different relations between the evaluator and the acting environment. Filesystem tasks use a lightweight local environment that can share the evaluation container and therefore run in Simple mode. GitHub and Notion tasks rely on remote external services that the platform does not need to host directly, so they also remain compatible with Simple mode. In contrast, Playwright and Postgres require platform-hosted service stacks and are routed through DinD mode, where each task receives an isolated inner Docker environment.

In our public artifact, BenchRouter supports all five MCPMark server families—Filesystem, GitHub, Notion, Playwright, and Postgres—

Benchmark	Class	Bridge LOC	YAML LOC	Dockerfile LOC
lmms-eval	Evaluator-only	192	12	12
τ^2-bench	Evaluator-only	253	44	18
BrowseComp	Evaluator-only	99	10	5
xbench	Evaluator-only	153	13	15
MCPMark	Co-resident / Remote external	134	43	135
Toolathlon	Hosted heavy env	241	25	130
WebArena	Hosted heavy env	367	16	37
SWE-bench Pro	Benchmark-owned exec	227	4	reused[§]
Upstream scorer LOC modified	**0 across all eight benchmarks**			

Table 3: Platform-specific code per integrated benchmark, ordered by environment class (Table 5). *Bridge LOC* counts the BenchRouter-side benchmark-specific Python files (entry scripts and per-benchmark bridge code); *YAML LOC* sums *benchmark.yaml* and all *task.yaml* files; *Dockerfile LOC* counts benchmark-specific Dockerfile lines. Across all eight benchmarks the bridge size has median 210 and mean 208. The last footer row records the single strongest invariant of the study: upstream scorer code is never modified. [§]SWE-bench Pro reuses upstream Dockerfile machinery and ships a task generator plus template task instead of a benchmark-specific Dockerfile.

through thin bridge code. The bridge injects model configuration, invokes MCPMark's native entry points, and normalizes the exported result file. Importantly, the MCPMark evaluator and scoring logic remain upstream rather than being reimplemented inside BenchRouter.

MCPMark provides clear fidelity check because it exposes both a public leaderboard and a benchmark-owned evaluation path. We therefore compare BenchRouter's routed results against the official MCPMark results under matched model and prompt settings across all five public server families. Table 4 reports the official MCPMark score, the BenchRouter-routed score, and their absolute gap. Across the five server families, the routed scores remain close to the official scores, suggesting that BenchRouter's execution boundary—model endpoint injection, container dispatch, and result collection—does not materially change benchmark behavior.

The intended point is architectural: by keeping MCPMark's own evaluator authoritative, BenchRouter avoids re-expressing server interaction, task dispatch, and scoring rules in platform-native code. This reduces the chance of introducing server-specific deviations while still allowing MCPMark to be executed through the same router interface as the other benchmarks.

5.3 SWE-bench Pro

SWE-bench Pro is heavy not because of a browser or service environment but because its benchmark-owned execution system is itself heavy: the evaluator builds or loads containerized repositories, applies patches, and runs tests. This stresses

Server	Official	BenchRouter	$\lvert\Delta_{\mathrm{BR}}\rvert$
Filesystem	32.50	33.33	0.83
GitHub	29.35	28.26	1.09
Notion	25.00	25.00	0.00
Playwright	27.00	26.00	1.00
Postgres	50.00	48.81	1.19

Table 4: MCPMark fidelity on the five public server families, evaluated with `claude-sonnet-4-5`. Scores use MCPMark's 0–100 success-rate scale; *Official* is the public leaderboard, *BenchRouter* is under matched prompt settings. BenchRouter leaves MCPMark's evaluator and scoring logic unchanged, so the residual gaps (max 1.2, mean 0.82 points) reflect run-to-run sampling variance.

DinD support, image preparation, and partial-result handling. Each task gets isolated inner Docker state, generated shards run independently under the Run/Job model, and the upstream evaluator remains the scoring authority. The 754-vs-227 LOC contrast against Inspect's SWE-bench port (already noted above) further illustrates this division in platform-versus-upstream ownership of execution steps.

6 Conclusion

For environment-heavy agent benchmarks the practical bottleneck is runtime integration. BenchRouter shows that routing benchmarks as opaque workloads is a workable alternative to rewriting their evaluators: across eight integrations the router-facing code stays small, upstream scorers remain untouched, and multi-host placement, shared GPU pools, and broader fidelity audits beyond MCPMark are the immediate next steps.

Limitations

BenchRouter's main limitation is empirical reach: the fidelity audit is concentrated on MCPMark because it is the only benchmark in our 8-benchmark sample that exposes both a public leaderboard and a benchmark-owned scorepath that we can match under identical model and prompt settings. Even with avg@4 multi-family coverage in Table 4, the audit therefore remains tied to MCPMark's official scorepath, and we do not generalize the closeness of routed scores to official scores beyond these operating points; extending it depends on the broader ecosystem publishing more matched scorepaths rather than on additional engineering on our side. Further boundary considerations on contract scope, DinD isolation, observability, scheduler-level fairness, and the remaining behavioral head-to-head against Inspect are discussed in Appendix A.9.

References

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. *arXiv preprint arXiv:2506.07982*.

Kaiyuan Chen, Yixin Ren, Yang Liu, Xiaobo Hu, Haotong Tian, Tianbao Xie, Fangfu Liu, Haoye Zhang, Hongzhang Liu, Yuan Gong, and 1 others. 2025. xbench: Tracking agents productivity scaling with profession-aligned real-world evaluations. *arXiv preprint arXiv:2506.13651*.

Suhwan Choi, Yunsung Lee, Yubeen Park, Chris Dongjoo Kim, Ranjay Krishna, Dieter Fox, and Youngjae Yu. 2026. vla-eval: A unified evaluation harness for vision-language-action models. *arXiv preprint arXiv:2603.13966*.

OpenCompass Contributors. 2023. Opencompass: A universal evaluation platform for foundation models. https://github.com/open-compass/opencompass.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Haodong Duan, Junming Yang, Yuxuan Qiao, Xinyu Fang, Lin Chen, Yuan Liu, Xiaoyi Dong, Yuhang Zang, Pan Zhang, Jiaqi Wang, and 1 others. 2024. Vlmevalkit: An open-source toolkit for evaluating large multi-modality models. In *Proceedings of the 32nd ACM international conference on multimedia*, pages 11198–11201.

Kelin Fu, Tianyu Liu, Zeyu Shang, Yingwei Ma, Jian Yang, Jiaheng Liu, and Kaigui Bian. 2025. Multi-docker-eval: Ashovel of the gold rush'benchmark on automatic environment building for software engineering. *arXiv preprint arXiv:2512.06915*.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, and 5 others. 2024. The language model evaluation harness.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Vaibhav Mehta, Lalit K Jain, Virginia Aglietti, Disha Jindal, Yuanzhu Peter Chen, and 1 others. 2025. Big-bench extra hard. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26473–26501.

Alexandre Lacoste, Nicolas Gontier, Oleh Shliazhko, Aman Jaiswal, Kusha Sareen, Shailesh Nanisetty, Joan Cabezas, Manuel Del Verme, Omar G Younis, Simone Baratta, and 1 others. 2026. Cube: A standard for unifying agent benchmarks. *arXiv preprint arXiv:2603.15798*.

Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, and 1 others. 2025. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. *arXiv preprint arXiv:2510.25726*.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, and 1 others. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2024. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pages 52989–53046.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng

Kong, and Junxian He. 2024. Agentboard: An analytical evaluation board of multi-turn llm agents. *Advances in neural information processing systems*, 37:74325–74362.

Chengyu Shen, Yanheng Hou, Minghui Pan, Runming He, Zhen Hao Wong, Meiyi Qiang, Zhou Liu, Hao Liang, Peichao Lai, Zeang Sheng, and 1 others. 2026. One-eval: An agentic system for automated and traceable llm evaluation. *arXiv preprint arXiv:2603.09821*.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, and 1 others. 2026. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*.

UK AI Security Institute, Arcadia Impact, and Vector Institute. 2024. *inspect_evals*: Collection of evals for Inspect AI. https://anonymous.4open.science/r/inspect_evals-57FD. GitHub repository.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. 2025. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*.

Zijian Wu, Xiangyan Liu, Xinyuan Zhang, Lingjun Chen, Fanqing Meng, Lingxiao Du, Yiran Zhao, Fanshi Zhang, Yaoqi Ye, Jiawei Wang, and 1 others. 2025. Mcpmark: A benchmark for stress-testing realistic and comprehensive mcp use. *arXiv preprint arXiv:2509.24002*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Frank Fangzheng Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, and 1 others. 2026. Theagentcompany: benchmarking llm agents on consequential real world tasks. *Advances in Neural Information Processing Systems*, 38.

Kaichen Zhang, Bo Li, Peiyuan Zhang, Fanyi Pu, Joshua Adrian Cahyono, Kairui Hu, Shuai Liu, Yuanhan Zhang, Jingkang Yang, Chunyuan Li, and Ziwei Liu. 2024. Lmms-eval: Reality check on the evaluation of large multimodal models. *Preprint*, arXiv:2407.12772.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.

A Reproducibility Details

BenchRouter's main text focuses on the contract and integration study. This appendix records the operational defaults behind the reported results, the full taxonomy table referenced in Section 5, the boundary cases deferred from the main text, and the small in-container runner that standardizes benchmark outputs.

A.1 Full runtime taxonomy

The main text replaces the taxonomy table with a short prose summary to save body pages. Table 5 gives the full per-benchmark assignment used in that summary.

Benchmark	Class	Acting environment	Mode
lmms-eval	Eval.-only	None (QA-only, no acting environment)	Simple
τ^2-bench	Eval.-only	None	Simple
BrowseComp	Eval.-only	None	Simple
xbench	Eval.-only	None	Simple
MCPMark (Filesystem)	Co-resident	Lightweight; shares the evaluation container	Simple
MCPMark (Notion)	Remote external	Remote service, not hosted by the platform	Simple
Toolathlon	Hosted heavy	Heavy service stack, platform-hosted	DinD
WebArena	Hosted heavy	Heavy browser-backed stack, platform-hosted	DinD
SWE-bench Pro	Benchmark-owned	Benchmark builds and runs its own containers	DinD

Table 5: Integration regimes covered by the selected benchmarks, organized by how the *acting environment* relates to the *evaluation container*. *Evaluator-only* benchmarks need only the evaluation container itself (typical QA). *Co-resident env* benchmarks need an environment so lightweight that it can share the evaluation container. *Remote external env* benchmarks rely on a remote service that the platform does not need to host. *Hosted heavy env* benchmarks require a hosted heavy environment stack. *Benchmark-owned exec* benchmarks bring their own heavy execution machinery. MCPMark spans co-resident and remote-external classes at the task-family level and therefore appears in two rows.

A.2 Execution defaults

Simple-mode containers default to 16 GB RAM and 4 CPU cores (`cpu_quota=400000`, `cpu_period=100000`), while DinD-mode containers default to 32 GB RAM. The maximum concurrent job count defaults to 8 and is bounded by a threading semaphore. Default per-task timeout is 7200 seconds. All limits are configurable per task in *task.yaml*.

A.3 Storage layout

The platform stores all state under a single *data_dir*: `benchmarks/` (registered benchmark archives), `environments/` (Dockerfiles and build metadata), `runs/` and `jobs/` (JSON-serialized state), and `results/` (per-job *result.json* and an append-only *index.jsonl*). On server restart, the scheduler reloads runs and jobs from disk; in-flight container state is reconciled at the Docker API.

A.4 Provenance, observability, and isolation

BenchRouter persists a per-run provenance record in `runs/`⟨`run_id`⟩`/` and the append-only `results/index.jsonl`. The record contains the benchmark identity, the SHA-256 of the registered benchmark archive, snapshots of the effective `task.yaml` and referenced `benchmark.yaml` files copied into the run directory at submit time, the container image tag and resolved image digest, the model endpoint URL and model identifier, and a UTC timestamp range for the run. When a benchmark exposes a seed (either as a `seed` field in `task.yaml` or via `BENCHROUTER_SEED` in `extra_env`), BenchRouter captures it as part of the run metadata rather than leaving it only in benchmark-local logs.

Current observability is intentionally coarse. The router tracks job-level state transitions, duration, exit code, partial/completed status, and the last 50 lines of container logs, while preserving the benchmark's exported `raw_metrics` verbatim in the standardized result file. It does not yet capture step-level traces, per-tool-call timings, or structured benchmark-internal events, so debugging still falls back to benchmark-local logs and reruns.

DinD isolation similarly follows an operational rather than a hardened-security model. Under Sysbox, nested containers run inside their own user-namespaced environment; where Sysbox is unavailable, the privileged fallback expands the trust boundary further. The current artifact therefore relies on Docker/Sysbox isolation primitives and deployment hygiene, but does not claim protection against adversarial benchmark code or container escape attempts.

A.5 Bridge integration patterns

Integrations also differ in how the bridge code is written, beyond the environment-class taxonomy. Which form a benchmark takes is determined by the integration surface it already exposes upstream, not by a choice the platform makes.

Pattern A: Bridge script. When the upstream benchmark already exposes a CLI entry point, the bridge is a short wrapper that translates BenchRouter's environment variables into native CLI flags, invokes the benchmark's own entry point, and writes *result.json*. This is the most common case (5/8: MCPMark, WebArena, BrowseComp, τ^2-bench, Toolathlon), reflecting that most modern agent benchmarks ship a CLI before a stable framework-facing API.

Pattern B: In-process patch. When no CLI entry point is convenient, the bridge keeps the upstream evaluator in process and patches only the model-call boundary. If the benchmark exposes a usable Python API, the bridge imports it (lmms-eval); if not, it patches the benchmark's own code path at the model call site (xbench, SWE-bench Pro). API-level patches are robust to upstream churn, while source-level patches can break on an upstream rename in the model client.

A.6 Failure handling

A job is marked FAILED on (i) container exit with non-zero code, (ii) timeout, (iii) missing *result.json*, or (iv) missing *overall* field. The last 50 lines of container logs are captured into the job record, and container cleanup runs in a `finally` block to avoid leaks.

A.7 Boundary cases and failure modes

The main text discusses only benchmark families that satisfy the current contract. Two nearby cases illustrate where the contract becomes strained. *OSWorld* (Xie et al., 2024) expects host-device passthrough (e.g. `/dev/kvm`) and VM-snapshot semantics that the current manifest does not expose. *The AgentCompany* (Xu et al., 2026) relies on agent state that persists across many work-day-length tasks, which the per-task container lifetime tears down. A `devices` or `persistent_volume` field in *task.yaml* would be a natural first extension point, but those additions may still be insufficient in practice because they do not by themselves solve the lifecycle and state-management assumptions of these benchmarks.

A.8 SDK runner

Inside every benchmark container, a small runner is responsible for the last normalization step. It reads the task manifest, executes the benchmark command, loads the raw result file, applies a single field remap, and writes the standardized record expected by the router:

```
read task.yaml
run command in /app/benchmark
load result_file (default result.json)
overall = raw[mapping.get("overall",
                          "overall")]
write {benchmark_name, model_name,
       overall, raw_metrics} to
       /app/results/result.json
```

This runner is the only platform code that executes inside a benchmark-owned container. The benchmark's own evaluator, scorer, and task flow remain otherwise unchanged. The runner remaps one benchmark-owned score to *overall*, but it does not compute composite metrics on the platform's behalf.

A.9 Scope and design boundaries

The main Limitations section names the one principal empirical limitation; the rest of the design's scope and trade-offs are discussed here.

Contract scope. The current contract is intentionally narrow. It is strongest when a benchmark can be expressed as endpoint-style inference plus one benchmark-owned *overall* metric. Richer interfaces such as tool/function calling, streaming outputs, multimodal request schemas, or non-OpenAI transport protocols would require typed optional fields or a pluggable transport layer. Multi-metric benchmarks remain benchmark-owned: BenchRouter records one authoritative *overall* field and preserves the rest as raw metrics rather than standardizing composite semantics itself.

Execution and security. DinD requires Sysbox or a privileged fallback, and the current implementation is single-host rather than a multi-host scheduler. GPU placement, device passthrough (e.g. `/dev/kvm`), and persistent cross-task state are therefore out of scope today. BenchRouter does not claim hardened sandboxing against adversarial benchmark code; it relies on container isolation primitives and a trusted deployment environment rather than VM-grade security guarantees.

Observability and scheduling. The opaque-workload design makes debugging coarser than in a platform-owned evaluator. BenchRouter records standardized outputs, job status, and container logs, but does not yet expose step-level traces, per-tool-call telemetry, or benchmark-internal health checks as first-class platform events. Scheduler-level fairness and global rate limiting across model endpoints are likewise delegated to the endpoint and the benchmark's own retry logic; a platform-side token bucket would require schema knowledge of each benchmark's request semantics and is left to future work.

Empirical proxies. The integration study is author-conducted, and Bridge LOC is only a static proxy for adaptation surface rather than a measure of person-hours or longitudinal maintenance burden under upstream churn. A quantitative behavioral head-to-head against Inspect on a matched task subset and identical model endpoint likewise remains future work; the present comparison is structural (Table 3 and the case-study contrasts) rather than score-based.