LLMContextBench: A Benchmark Tool for Evaluating Context Provisioning Strategies in LLM-Based Systems

Michel Vasconcelos
michel.vasconcelos@gmail.com
Universidade de Fortaleza
Fortaleza, CE, Brazil

Nabor C. Mendonça
nabor@unifor.br
Universidade de Fortaleza
Fortaleza, CE, Brazil

Abstract

Large Language Model (LLM)-based systems increasingly depend on external context, such as documents, structured records, source code, logs, and tool outputs. However, developers and researchers still lack practical tooling to evaluate how alternative context provisioning strategies affect answer quality, token consumption, latency, call behavior, and observability. This paper presents LLMContextBench, a command-line benchmark tool for planning, executing, evaluating, and exporting reproducible experiments on context provisioning in LLM-based systems. The tool separates datasets, strategies, models, evaluation, and artifacts through a common lifecycle, and supports inline context, benchmark-controlled local function calling, local Model Context Protocol (MCP) execution, and provider-managed remote MCP execution. We illustrate the tool with two datasets: Lattes, based on semi-structured academic curricula, and RepoQA, based on software repository retrieval tasks. The experiments compare prompt-based, operation-mediated, and protocol-based access using LLM-as-a-judge assessment for Lattes and deterministic scoring for RepoQA. Results show domain-dependent trade-offs: in Lattes, inline JSON achieves the highest majority accuracy but high token cost, while operation-mediated strategies reduce tokens; in RepoQA, inline code and JSON achieve perfect retrieval, whereas operation-mediated strategies depend on operation precision and output-format control. These findings show how LLMContextBench supports evidence-based decisions about context provisioning architectures.

Demo video: https://github.com/michelav/llmcontextbench/blob/main/demo-video.md

Keywords

Large Language Models, Context Provisioning, Model Context Protocol, Benchmark Tools, Empirical Software Engineering

1 Introduction

Large Language Models (LLMs) are increasingly embedded in software systems for question answering, information extraction, summarization, and decision support [29]. In these systems, output quality depends not only on the model, but also on how the application exposes task-relevant context such as documents, structured records, source code, logs, or tool outputs. Context provisioning — how an application supplies external, task-relevant information to the model —has therefore become an increasingly important software engineering concern in LLM-based systems [14].

Several mechanisms can be used to provide external context. Applications may serialize context directly into the prompt, retrieve selected evidence before generation, expose local functions, or connect the model to external tools through the Model Context Protocol (MCP) [1, 7, 9, 16, 17]. These mechanisms differ in visibility, token cost, latency, control flow, deployment complexity, and observability [8, 13, 23, 26]. Consequently, developers need experimental support for comparing context provisioning alternatives under the same tasks, datasets, models, and evaluation procedures.

This paper presents *LLMContextBench*, a command-line benchmark tool for evaluating context provisioning strategies as system-level architectural decisions. The tool supports a reproducible lifecycle that plans trials, executes them with alternative strategies, evaluates generated responses with configurable judges or scorers, records traces and metrics, and exports analysis-ready artifacts. Its dataset abstraction separates domain-specific context access from the benchmark engine, allowing datasets to expose context as files, structured records, local operations, MCP tools, or remote services. LLMContextBench primarily targets software developers and researchers who reason about solution architecture, rather than end users who are typically limited to writing prompts or submitting questions.

We demonstrate the tool with two datasets. The Lattes dataset is based on large semi-structured academic curricula from the Brazilian Lattes Platform [3]; RepoQA is based on software repository question-answering tasks over code artifacts [12]. These domains exercise different evaluation needs: Lattes uses LLM-as-a-judge assessment for answers over semi-structured curricula, whereas RepoQA uses deterministic scoring for source-code retrieval. Initial experimental results show that no context provisioning strategy is uniformly best and that the relevant design lever may vary by domain: data representation is especially important in Lattes, while operation design and precision are central in RepoQA.

The remainder of this paper is organized as follows. Section 2 presents the design goals, core concepts, architecture, supported strategies, and lifecycle artifacts of LLMContextBench. Section 3 describes dataset integration, experiment definition, and benchmark execution. Section 4 reports the Lattes and RepoQA results and discusses their implications. Section 5 reviews related work, and Section 6 concludes the paper.

2 LLMContextBench

LLMContextBench is a benchmark tool for assessing context provisioning architectures in LLM-based systems. The tool is motivated by the observation that, in many LLM-based applications, task-relevant information is not contained in the model itself, but in external artifacts such as documents, structured records, source code, logs, repositories, or domain-specific services. In this setting, software engineers must decide how such context is exposed to the model: it may be serialized directly into the prompt, accessed through local operations, served through a protocol such as MCP, or

delegated to a remote context service. LLMContextBench supports controlled experiments over these alternatives, allowing users to compare their effects on response quality, token usage, latency, tool behavior, and observability.

The tool is implemented as a command-line workflow around three design principles. First, experiments are declarative: users describe the dataset, tasks, instances, models, context provisioning strategies, and evaluation configuration in an experiment file. Second, the workflow is artifact-oriented: the tool expands the experiment into planned trials, executes them, evaluates the produced responses, records traces and metrics, and exports analysis-ready results. Third, the dataset layer is extensible: domain-specific datasets are incorporated through packages and providers that map their native artifacts, tasks, instances, evidence, and optional operations to the common benchmark contract.

2.1 Design Goals and Core Concepts

LLMContextBench was designed to support reproducible, inspectable, and extensible experiments over context provisioning alternatives. Its first goal is *comparability*: the same tasks, instances, models, and evaluation settings should be executable across different strategies. Its second goal is *observability*: the benchmark should collect enough information to analyze not only the final answer, but also token usage, latency, tool calls, MCP calls, judge votes, and errors. Its third goal is *extensibility*: a new dataset should be added by implementing a provider rather than by modifying the benchmark lifecycle. Its fourth goal is *artifact orientation*: every phase should consume and produce explicit files so that experiments can be inspected, resumed, filtered, and analyzed outside the tool.

Figure 1 summarizes the main concepts used by the tool. An *experiment* declares the design space, output location, and complete assessment workflow. It references a *dataset*, selects the *instances* and *tasks* to be evaluated, defines the models and strategies to compare, and configures how generated responses should be evaluated. A *dataset* is a versioned benchmark input package that provides tasks, instances, context representations, optional operations, evaluation evidence, and optional scoring logic. An *instance* is a concrete contextual unit. In the Lattes dataset, for example, an instance is a curriculum identifier associated with files such as HTML, parsed JSON, and evidence blocks. In RepoQA, an instance represents a software repository context, such as a repository snapshot or changeset. A *task* describes what the model should do over an instance, such as answering a question, searching for information, retrieving a code element, or performing a domain-specific analysis.

The central execution unit is a *trial*. A trial is produced during planning and contains all information required to execute one experimental condition: dataset provenance, instance, task statement, model and parameters, provisioning strategy, context representation, and repetition index. Conceptually, each trial binds one instance, one task, one model, one strategy, one representation, and one repetition. During execution, each trial produces a *response*, which stores the model output together with execution status, token usage, timing, compact metrics, and trace references. Each response is then assessed through an *evaluation*, which stores judge or scorer outcomes according to the validation method configured for the task. The validation may be performed by LLM judges, by a deterministic scorer, or by another dataset-specific evaluator. Finally, execution and evaluation can emit *traces*, which record model calls, operation calls, raw outputs, timing, token usage, errors, and other details needed for later analysis.

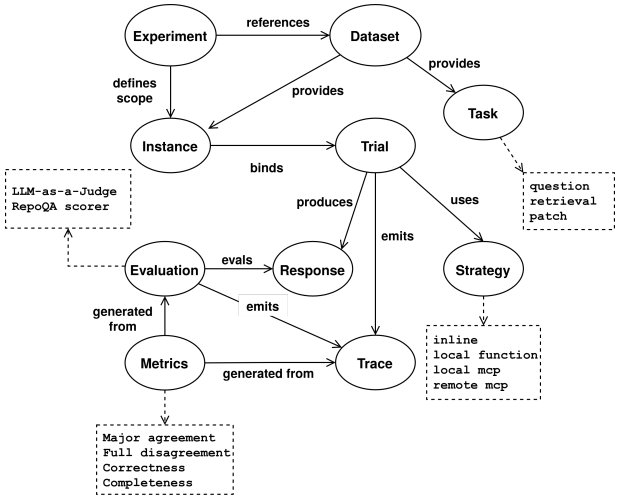

Figure 1: Conceptual domain model of LLMContextBench.

Table 1: Main architectural structures of LLMContextBench.

Structure	Main elements	Responsibility
UI / CLI Commands	commands, options, selectors	Invokes benchmark phases and filters trials, responses, and evaluations.
Benchmark Core	experiment model, planner, execution and evaluation engines, serializers	Expands experiments, coordinates execution and evaluation, records provenance, and persists artifacts.
Dataset Provider	resolver, task and instance catalogs, context, evidence, operations	Exposes dataset identity, tasks, instances, context, evidence, oracles, and operations through a common contract.
Strategy Layer	inline, local-function, local-MCP, remote-MCP strategies	Defines how context reaches the model: prompt injection, local operations, or protocol-mediated access.
AI / LLM Engine	model adapters, tool and MCP runtimes, usage accounting, raw responses	Handles model calls, prompt/tool formatting, token accounting, continuations, and raw responses.

2.2 Architecture Overview

The architecture of LLMContextBench is organized around five structures: UI/CLI Commands, Benchmark Core, Dataset Provider, Strategy Layer, and AI/LLM Engine. Table 1 summarizes their main elements and responsibilities. Together, these structures separate user interaction, benchmark lifecycle control, dataset access, context-provisioning decisions, and model/tool execution, keeping datasets, models, strategies, and artifacts independently inspectable.

Beyond these per-structure responsibilities, the defining property is separation of concerns: the Strategy Layer varies how context reaches the model—inline prompt injection, local operations, or protocol-mediated access—without changing dataset providers or model adapters. At run time, the UI/CLI Commands invoke the Benchmark Core, which resolves the dataset and strategy for each trial; the AI/LLM Engine performs the model interaction; and the core persists responses, evaluations, metrics, and traces as explicit

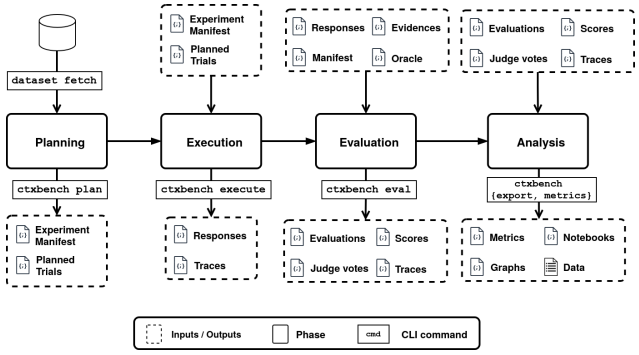

Figure 2: LLMContextBench workflow.

artifacts, enabling inspection, partial re-execution, reproducibility, and downstream analysis.

2.3 Context Provisioning Strategies

A core goal of LLMContextBench is to compare context provisioning architectures under the same lifecycle. The tool represents these architectures as *strategies*, each defining how the model accesses task-relevant context.

The `inline` strategy serializes a provider-supplied artifact, such as cleaned HTML, parsed JSON, code bundles, or repository context, directly into the prompt with the task. This strategy is simple and gives the model broad context visibility, but its token cost and latency scale with the size of the serialized artifact.

The remaining strategies are *operation-mediated*: the model receives access to operations that expose context on demand. In `local_function`, these operations are benchmark-controlled read-only functions. In `local_mcp`, the benchmark still controls the interaction loop, but operations are exposed through a local MCP runtime and client/server boundary. In `remote_mcp`, context access is delegated to a remote MCP server and provider-native tool support, approximating a deployed external context service but potentially increasing latency and reducing direct observability.

Together, these strategies compare both context representations and architectural boundaries. Inline versus operation-mediated execution measures the effect of moving context from prompts to explicit operations; `local_function` versus `local_mcp` isolates the local MCP boundary; and `local_mcp` versus `remote_mcp` captures the shift from benchmark-controlled execution to provider-mediated remote access.

2.4 Benchmark Workflow

LLMContextBench follows the workflow shown in Figure 2. The main phases are planning, execution, evaluation, and analysis, preceded when necessary by a dataset-fetch step. The workflow is intentionally artifact-oriented: each phase consumes explicit inputs, produces explicit outputs, and can therefore be inspected, resumed, filtered, or re-executed independently.

The workflow is exposed through a small set of command-line entry points. Dataset packages can be materialized or inspected with `ctxbench dataset fetch` and `ctxbench dataset inspect`. The benchmark lifecycle is then executed with `ctxbench plan`, `ctxbench execute`, `ctxbench eval`, and `ctxbench export`, while `ctxbench status` summarizes progress and pending work. These commands are not independent tools, but phase-specific views over the same lifecycle: each command reads artifacts produced by previous phases and writes the canonical inputs for subsequent phases.

Following Figure 2, planning (`ctxbench plan`) resolves the dataset reference, validates capabilities, and expands the experiment into an experiment manifest and a planned-trials artifact—the canonical input for generation. Execution (`ctxbench execute`) consumes the planned trials, applies optional selectors (model, provider, instance, task, strategy, format, repetition, or trial), and sends each trial to the configured model, producing response artifacts and, when enabled, execution traces. Evaluation (`ctxbench eval`) scores responses against dataset evidence or oracles and emits aggregate evaluations together with judge-level votes that support disagreement, majority, and unanimity analyses. Finally, the analysis phase derives metric summaries and tabular data through `ctxbench metrics` and `ctxbench export`, which feed the notebooks and graphs used for the reported results.

3 Using and Extending LLMContextBench

This section illustrates how a user works with LLMContextBench to design, execute and analyze experiments.

3.1 Adding a Dataset

A dataset is added to LLMContextBench by implementing a provider that maps a domain-specific source to the benchmark contract: dataset identity, tasks, instances, context representations, evidence, optional oracles, operations, and capability metadata. The provider hides physical storage and access details from the benchmark core. Internally, a dataset may consist of local files, generated artifacts, repositories, database records, or API-accessible resources. Externally, it exposes stable methods for listing identifiers, loading task definitions, retrieving context and evidence, reporting capabilities, and providing provenance information such as identity, version, and origin. This separation allows different datasets to share the same planning, execution, evaluation, and export workflow without introducing domain-specific assumptions into the core.

Implementing a provider typically requires defining metadata, task catalogs, and task-instance bindings; mapping context representations to concrete artifacts or access mechanisms; defining evidence, oracles, scoring inputs, or judge-visible resources; and, when supported, exposing local operations or MCP-compatible tools. Declared capabilities determine which strategies and evaluation methods apply. Thus, execution and evaluation remain separate but connected through the dataset contract: models may answer through inline context, local functions, local MCP, or remote MCP, while judges or scorers assess responses against provider-supplied evidence or oracles.

We illustrate the dataset provider implementation with two datasets: Lattes and RepoQA. Table 2 maps the common benchmark concepts to concrete structures in each dataset.

Lattes. Lattes is the Brazilian national curriculum platform used to register academic trajectories, including education, publications, projects, supervisions, experience, and scholarly production. Prior

Table 2: Mapping between benchmark concepts and concrete dataset structures.

Concept	Lattes	RepoQA
Dataset	Set of Lattes curricula.	Repository search-and-retrieve cases.
Instance	One Lattes curriculum.	One case: repository, needle, and context size.
Task	Question over a curriculum.	Retrieval task based on a symbol description.
Context	Clean HTML, parsed JSON, and evidence blocks.	Snapshots, compact contexts, metadata, and evidence bundles.
Evidence	Blocks and accepted values per task–curriculum pair.	Needle-specific repository evidence and oracle metadata.
Operations	Resource operations over profile, education, publications, projects, and outputs.	Repository operations for reading files and searching for code symbols.

work has already used Lattes curricula for LLM-supported question answering, notably LattesRex, a chatbot that combines classification, retrieval, and generation over semi-structured Lattes documents [4]. Here, Lattes is used instead as a dataset for comparing context provisioning strategies. A curriculum is a rich semi-structured document, making Lattes useful for evaluating strategies that must locate evidence across sections, apply temporal filters, aggregate records, or combine multiple evidence fragments.

In LLMContextBench, a Lattes instance corresponds to one curriculum. Each instance stores the original HTML page, cleaned HTML, parsed JSON, and block-level evidence. Inline strategies use the cleaned HTML or parsed JSON artifacts, whereas operation-mediated strategies use JSON as the backing source for local functions and MCP tools. Task-instance bindings provide parameters, accepted answers, themes, context references, and evaluation blocks, keeping curriculum-specific evidence outside the benchmark core.

The Lattes provider exposes resource-oriented operations over profile, expertise, education, projects, supervisions, experience, academic activities, publications, technical output, and artistic output. Most operations receive an instance identifier and, when relevant, temporal filters such as `start_year` and `end_year`. The current baseline uses five curricula and twelve tasks covering factual lookup, temporal filtering, aggregation, ranking, matching, weak-data cases, cross-block reasoning, and inferential synthesis.

RepoQA. RepoQA is an independent external dataset integrated into LLMContextBench through a dedicated provider. RepoQA benchmarks code understanding in long contexts [12]. Its Search Needle Function task asks a model to retrieve a target function from a repository-level code context using a natural-language description rather than the function name. Unlike Lattes, RepoQA uses software repository evidence composed of source files, dependency-ordered code, function descriptions, and hidden target functions, with answers scored against ground truth using a code-oriented metric.

We adapt RepoQA as an LLMContextBench dataset package rather than as a standalone runner. The package follows the same contract used by Lattes: dataset manifest, task catalog, task-instance bindings, context artifacts, evidence blocks, and hidden oracle artifacts. A RepoQA instance combines a base needle with a requested context size, allowing the same repository and target function to generate multiple instances under different code-context budgets.

The adapted task asks the model to reproduce the exact described function from the provided code context. Model-facing artifacts include a text code context, a structured JSON projection, and evaluation blocks; native RepoQA prompts and contexts are retained for reproducibility, while target functions and metadata remain hidden oracles. Inline strategies receive code context directly, operation-mediated strategies can use repository operations when exposed, and evaluation can apply RepoQA-specific similarity thresholds and comment-handling policies. Thus, RepoQA complements Lattes by stressing source-code retrieval and oracle-based scoring while preserving the shared lifecycle of planning, execution, evaluation, and export.

```
{
  "id": "experiment_001",
  "name": "Experiment example",
  "dataset": {
    "id": "ctxbench/lattes",
    "version": "2026-04-28",
    "root": "datasets/lattes"
  },
  "scope": {
    "instances": ["3457219624656691", "3957046121364560"],
    "tasks": ["q_phd", "q_field", "q_coauth"]
  },
  "models": {
    "gpt1": {
      "provider": "openai",
      "name": "gpt-5.4-nano"
    },
    "judge-gemini": {
      "provider": "google",
      "name": "gemini-2.5-flash-lite"
    }
  },
  "factors": {
    "model": ["gpt1"],
    "strategy": ["inline", "local_function", "local_mcp"],
    "format": ["html", "json"]
  },
  "params": {
    "mcp_server": {
      "server_url": "https://example.org/mcp",
      "auth_token": "${LATTES_MCP_TOKEN}"
    }
  },
  "evaluation": {
    "enabled": true,
    "judges": ["judge-gemini"]
  },
  "trace": {
    "enabled": true,
    "writeFiles": true
  }
}
```

Listing 1: Example of an LLMContextBench experiment file.

3.2 Defining an Experiment

An experiment file instantiates the conceptual model introduced in Section 2.1. Listing 1 shows a simplified specification. Lines 1–12 identify the experiment, select the dataset package and provenance, and restrict the scope to selected instances and tasks. Lines 13–22 declare model aliases, including a response-generation model and a judge model used later by the evaluation phase. Empty scope lists may be used to include all available instances or tasks, while explicit lists support development, debugging, and controlled experimental designs.

```
# traces/queries/d004908e6c.json  (abridged event stream)
model.generate  (2415 ms)      -> decides to call an operation
  mcp.tool_call    get_expertise(lattes_id="<id>")
  mcp.tool_result  is_error=false  (7 ms)
model.generate  (2600 ms)      -> produces the final answer
strategy.local_function.execute ; engine.execute

# recorded outcome and metrics (answers.jsonl / evals.jsonl)
status   = success
model    = gpt-5.4-nano     strategy = local_function
format   = json
calls    = modelCalls:2  functionCalls:1  steps:2
tokens   = 2018 in / 78 out / 2096 total
duration = 5026 ms
outcome  = correctness:meets  completeness:meets  (3/3 judges)
```

Listing 2: Abridged execution trace and recorded metrics for one operation-mediated run.

The experiment definition is factor-based. Lines 23–27 define the factors combined by the planner: model aliases, context provisioning strategies, and context formats. This design allows users to compare models, strategies, and representations while holding the dataset and task set fixed, or to keep a strategy fixed and compare datasets or domains.

Some options must be enabled at design time because they determine which phases and artifacts the benchmark will produce. Lines 28–33 configure parameters needed by strategy-specific execution, such as the remote MCP server. Lines 34–37 enable evaluation and select the judge model; without this block, response generation can still run, but judge-based evaluation artifacts are not planned. Lines 38–41 enable trace recording and file persistence; without tracing, later analysis cannot inspect model calls, operation calls, and raw provider responses in the same level of detail. The experiment file therefore declares both the design space and the enabled benchmark features, while the planner materializes executable combinations according to provider and strategy capabilities. In the Lattes baseline, for example, inline trials use both `html` and `json`, whereas operation-mediated strategies use JSON-backed operations.

Complementing the input specification, Listing 2 shows an execution trace persisted for a single `local_function` run. It highlights the observability the tool records for operation-mediated strategies: the model requests an operation, receives context on demand, and answers, with per-run status, token counts, and evaluated outcome persisted as explicit artifacts.

4 Experiments and Preliminary Results

This section reports initial experiments with the Lattes and RepoQA datasets. The goal is not to provide a definitive ranking of context provisioning strategies, but to show how LLMContextBench exposes quality, cost, latency, call behavior, and observability trade-offs across domains.

Experimental Setup The Lattes experiment compares inline HTML, inline JSON, local function calling, local MCP, and remote MCP. It uses five curricula, twelve tasks, four generation models, and one repetition, producing 1,200 response-generation trials and up to 3,600 judge votes. Its metrics include majority and unanimous correctness, execution tokens, normalized token cost, median duration, and observed calls.

Table 3: Effectiveness, token cost, latency, and observed calls by context provisioning configuration.

(a) Lattes dataset.

Config.	N	Maj.	Unan.	Tok./T	Tok./M	Tok./U	Sec.	Calls
I-HTML	240	42.1	26.3	79.3k	188.4k	302.0k	4.09	0.00
I-JSON	240	45.8	24.6	70.6k	153.9k	287.0k	4.01	0.00
Func.	240	35.0	18.8	11.7k	33.5k	62.6k	4.61	1.23
L-MCP	240	35.4	17.5	12.6k	35.4k	71.7k	4.45	1.29
R-MCP	240	35.8	14.6	9.0k	25.1k	61.7k	5.99	1.15

(b) RepoQA dataset.

Config.	N	Hit	Pass	Sim.	Tok./T	Tok./P	Sec.	Calls
I-Code	45	100.0	100.0	1.00	8.8k	8.8k	3.00	0.00
I-JSON	45	100.0	100.0	1.00	21.5k	21.5k	3.00	0.00
Func.	45	91.1	91.1	0.92	9.6k	10.5k	6.00	1.89
L-MCP	45	88.9	88.9	0.90	8.6k	9.6k	6.00	1.69

Note. I-* denotes inline context; Func., L-MCP, and R-MCP denote local functions, local MCP, and remote MCP. Maj./Unan. are Lattes judge agreement metrics; Hit/Pass/Sim. are RepoQA match, pass, and similarity metrics. Tok./T is median tokens per trial; Tok./M, Tok./U, and Tok./P normalize tokens by successful responses; Sec. is median seconds; Calls is mean observed calls.

The RepoQA experiment compares inline code, inline JSON, local function calling with JSON, and local MCP with JSON. It covers Python, Java, TypeScript, and Rust, three context sizes (4k, 8k, and 16k), four generation models, and one repetition. RepoQA uses deterministic scoring, so hit rate, pass rate, mean similarity, token cost, latency, and calls replace judge-based agreement metrics. Table 3 reports the main results for both datasets.

Quantitative Results In Lattes, inline JSON achieved the highest majority accuracy (45.8%), followed by inline HTML (42.1%). Operation-mediated configurations were lower and close to each other, ranging from 35.0% to 35.8%. However, they greatly reduced token use: inline JSON consumed 70.6k tokens per trial, whereas local function calling, local MCP, and remote MCP consumed 11.7k, 12.6k, and 9.0k, respectively. Remote MCP was the most token-efficient configuration, but it also had the highest median latency (5.99 seconds) and less direct observability.

In RepoQA, both inline configurations reached 100.0% hit and pass rates with mean similarity 1.00. Local function calling and local MCP reached 91.1% and 88.9% pass rates, with mean similarities of 0.92 and 0.90. Inline code was both effective and efficient, requiring 8.8k tokens per trial, while inline JSON was much more expensive at 21.5k. Local MCP had a similar token cost to inline code (8.6k), but operation-mediated configurations doubled median latency to 6.00 seconds and introduced 1.69–1.89 calls per trial. These results suggest that, for RepoQA, operation precision and output-format control are central to performance.

Discussion The experimental results show how LLMContextBench supports context-aware architectural decisions across distinct domains. In the Lattes baseline, inline JSON achieved the highest aggregate quality because it gave the model broad curriculum visibility, but this came with high token consumption. In RepoQA, inline code and inline JSON achieved perfect retrieval, but

with different token profiles: inline code was substantially more efficient, whereas inline JSON expanded the context representation without improving quality. These results illustrate that inline context can be highly effective, but its cost-benefit ratio depends on the representation and domain.

The strategy comparisons also clarify where costs arise. In Lattes, local function calling and local MCP produced similar results, suggesting that the MCP boundary itself was not the dominant factor in this setting. Remote MCP reduced measured token cost but increased latency and reduced direct observability. In RepoQA, operation-mediated strategies remained competitive but did not match inline execution, indicating that repository operations must expose sufficiently precise navigation and retrieval mechanisms. Across both datasets, operation-mediated strategies reduced or controlled prompt size, but their effectiveness depended on the structure, granularity, and formatting of operation outputs.

The contrast between Lattes and RepoQA indicates that these trade-offs are domain-dependent. For Lattes, data representation and evidence structure strongly affect performance over semi-structured curricula, especially in tasks involving counting, temporal filtering, entity matching, and supervision-related information. For RepoQA, where the target is a precise code element, operation design and output-format control are central to accurate code reproduction. Thus, some domains may benefit more from better representations, while others require more precise operations.

Limitations. This baseline uses a single repetition per cell, so small differences should be read as preliminary evidence rather than definitive rankings. The persisted artifacts still support robustness analysis without re-execution—a block bootstrap over them estimates the inline versus operation-mediated accuracy gap at +8.5 points (95% CI [+1.9, +15.5])—and are available in the replication package [24].

5 Related Work

LLMContextBench relates first to broader efforts on reproducible and multi-dimensional evaluation of LLM-based systems. HELM emphasizes transparent multi-metric evaluation [11], while lm-evaluation-harness supports reproducible evaluation across model / task combinations [2]. These frameworks motivate LLMContextBench's artifact-oriented design, but they are not specialized for comparing context provisioning architectures. LLMContextBench focuses on how task-relevant context is exposed to the model, allowing the same dataset, tasks, models, and evaluation procedures to be executed with inline context, local functions, local MCP, or remote MCP.

LLMContextBench is also related to context engineering for LLM-based systems, including retrieval, context processing, memory, tool-integrated reasoning, and agentic workflows [14]. RAG combines parametric generation with retrieved non-parametric memory [9], and RAG evaluation studies retrieval relevance, faithfulness, and answer quality [5, 28]. Long-context studies further show that placing all evidence in the prompt does not ensure effective use of information [13, 26]. LLMContextBench is complementary to this work because it treats context provisioning as an architectural decision: context may be serialized into prompts, exposed through local operations, mediated by local MCP, or delegated to a remote MCP service.

Finally, several methods and benchmarks evaluate LLM tool use. ReAct interleaves reasoning and actions [27], while Toolformer studies learned API use [21]. API-Bank evaluates tool-augmented LLMs through API-call prediction and execution [10]. Gorilla focuses on connecting LLMs to large collections of APIs [19], ToolBench evaluates instruction-following and multi-step tool use over real-world APIs [20], and the Berkeley Function Calling Leaderboard evaluates function-calling behavior across models [18]. MCP-focused benchmarks extend this line by evaluating agents in MCP environments: MCP-RADAR provides a multi-dimensional benchmark for tool-use capabilities with MCP [6], MCP-Bench benchmarks tool-using agents with MCP [25], LiveMCPBench evaluates whether agents can navigate large collections of MCP tools [15], and MCPGauge studies whether MCP-augmented LLMs help or hinder agent performance [22]. LLMContextBench is complementary: rather than evaluating tool-use ability alone, it compares the software architectures that expose, control, observe, or delegate context access, including local functions, local MCP, and provider-managed remote MCP.

6 Conclusion

This paper presented LLMContextBench, a domain-agnostic command-line benchmark tool for evaluating context provisioning strategies in LLM-based systems. We illustrated the tool experimentally using two public datasets: Lattes and RepoQA. The results show that no strategy is universally best. In Lattes, inline JSON produced the highest accuracy, but at a substantially higher token cost than operation-mediated strategies. In RepoQA, inline code and inline JSON achieved perfect retrieval, but inline code was considerably more token-efficient, while operation-mediated strategies depended more strongly on operation precision and output-format control. These findings reinforce the need to treat context provisioning as an architectural decision that can be measured and refined experimentally.

Future work will evaluate additional datasets and repeated executions per model, refine operation design for different context types, support additional provider APIs, and improve evaluation metrics, cross-dataset comparability, trace inspection, and human auditing. We also plan to investigate more accessible experiment configuration interfaces, dynamic adaptation of multiple context provisioning strategies, and broader comparisons with related context- and tool-use benchmarks.

Artifact Availability

LLMContextBench is available at https://github.com/michelav/llmcontextbench under the MIT License, and a replication package (datasets and analysis scripts) is archived on Zenodo at https://doi.org/10.5281/zenodo.22118922.

Acknowledgements

Generative AI tools were used to assist in structuring and revising this manuscript as well as generating code for data collection and analysis.

References

[1] Anthropic. 2024. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol. Accessed: 2026-04-30.

[2] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, et al. 2024. Lessons from the Trenches on Reproducible Evaluation of Language Models. *arXiv preprint arXiv:2405.14782* (2024).

[3] Conselho Nacional de Desenvolvimento Científico e Tecnológico. 2023. Plataforma Lattes. https://www.gov.br/cnpq/pt-br/acesso-a-informacao/acoes-e-programas/plataforma-lattes. Accessed: 2026-04-30.

[4] Lucas Darcio et al. 2025. LattesRex: Building ChatBots for Semi-Structured Documents. In *Simpósio Brasileiro de Tecnologia da Informação e da Linguagem Humana (STIL)*. SBC, 125–136.

[5] Shahul Es, Jithin James, Luis Espinosa Anke, and Steven Schockaert. 2024. RAGAs: Automated Evaluation of Retrieval Augmented Generation. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics, 150–158. doi:10.18653/v1/2024.eacl-demo.16

[6] Xuanqi Gao et al. 2025. MCP-RADAR: A Multi-Dimensional Benchmark for Evaluating Tool Use Capabilities in Large Language Models. *arXiv preprint arXiv:2505.16700* (2025).

[7] Google. 2026. Function Calling with the Gemini API. https://ai.google.dev/gemini-api/docs/function-calling. Accessed: 2026-04-30.

[8] Xinyi Hou et al. 2025. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. *ACM Transactions on Software Engineering and Methodology* (2025).

[9] Patrick Lewis et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, Vol. 33. 9459–9474.

[10] Minghao Li et al. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In *Proc. of the Conference on Empirical Methods in Natural Language Processing*. 3102–3116.

[11] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic Evaluation of Language Models. *arXiv preprint arXiv:2211.09110* (2022).

[12] J. Liu and et al. 2024. RepoQA: Evaluating Long Context Code Understanding. *arXiv preprint arXiv:2406.06025* (2024).

[13] Nelson F. Liu et al. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[14] Lingrui Mei et al. 2025. A Survey of Context Engineering for Large Language Models. *arXiv preprint arXiv:2507.13334* (2025).

[15] Guozhao Mo, Wenliang Zhong, Jiawei Chen, Qianhao Yuan, Xuanang Chen, Yaojie Lu, Hongyu Lin, Ben He, Xianpei Han, and Le Sun. 2025. LiveMCPBench: Can Agents Navigate an Ocean of MCP Tools? *arXiv preprint arXiv:2508.01780* (2025).

[16] Model Context Protocol. 2025. Model Context Protocol Specification. https://modelcontextprotocol.io/specification/2025-11-25. Accessed: 2026-04-30.

[17] OpenAI. 2025. Function Calling in the OpenAI API. https://developers.openai.com/api/docs/guides/function-calling. Accessed: 2026-04-30.

[18] Shishir G Patil et al. 2025. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Proc. of the 42nd International Conference on Machine Learning (ICML)*.

[19] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large Language Model Connected with Massive APIs. *Advances in Neural Information Processing Systems* 37 (2024), 126544–126565.

[20] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *Proceedings of the 12th International Conference on Learning Representations*. https://openreview.net/forum?id=dHng2O0Jjr

[21] Timo Schick et al. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, Vol. 36.

[22] Wei Song, Haonan Zhong, Ziqi Ding, Jingling Xue, and Yuekang Li. 2025. Help or Hurdle? Rethinking Model Context Protocol-Augmented Large Language Models. *arXiv preprint arXiv:2508.12566* (2025).

[23] Thoughtworks. 2025. The Model Context Protocol: Getting Beneath the Hype. https://www.thoughtworks.com/en-br/insights/blog/generative-ai/model-context-protocol-beneath-hype. Accessed: 2026-04-30.

[24] Michel Vasconcelos and Nabor C. Mendonça. 2026. LLMContextBench Replication Package. doi:10.5281/zenodo.22118922

[25] Zhenting Wang et al. 2025. MCP-Bench: Benchmarking Tool-Using LLM Agents with Model Context Protocol. *arXiv preprint arXiv:2508.20453* (2025).

[26] Peng Xu et al. 2024. Retrieval Meets Long Context Large Language Models. In *Proc. of the 12th International Conference on Learning Representations (ICLR)*.

[27] Shunyu Yao et al. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proc. of the 11th International Conference on Learning Representations (ICLR)*.

[28] Hao Yu et al. 2024. Evaluation of Retrieval-Augmented Generation: A Survey. In *CCF Conference on Big Data*. Springer, 102–120.

[29] Murong Yue. 2025. A Survey of Large Language Model Agents for Question Answering. *arXiv preprint arXiv:2503.19213* (2025).