Diploma Project

BenchAgent: a Research Assistant for Benchmarking of Edge Infrastructures

Arina Kostina

University of Cyprus
Department of Computer Science

Department of Computer Science

May 2026

UNIVERSITY OF CYPRUS

Faculty of Pure and Applied Sciences

Department of Computer Science

BenchAgent: a Research Assistant for Benchmarking of Edge Infrastructures

Arina Kostina

Advisor:

Professor Marios Dikaiakos

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content.

May 2026

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Professor Marios Dikaiakos, for his invaluable guidance, encouragement, and support throughout my thesis, and research journey. Being under his supervision has significantly contributed to both my academic and personal development, helping me navigate challenges, and grow as a researcher. I am also deeply thankful to PhD candidate Joanna Georgiou for her continuous support, insightful feedback, and valuable guidance throughout my thesis.

I would also like to express my heartfelt gratitude to my family and friends for their love, encouragement, and unwavering support throughout this journey. Their belief in me and continuous encouragement helped me remain focused and determined during my years of study at the University of Cyprus. This work would not have been possible without them.

ABSTRACT

Evaluating data intensive applications on micro-datacenters has become increasingly essential as Edge computing and 5G networks demand low-latency, resource-efficient processing at scale. Benchmarking frameworks, such as BenchPilot, address these needs with declarative experiment specification, automated deployment, and unified monitoring. However, effective use of these systems requires significant operational expertise and close familiarity with its workflows, configuration formats, and documentation, posing a practical barrier for users whose primary focus is experiment design rather than infrastructure management. Recent advances in large language models (LLMs) and agentic systems offer a compelling opportunity to reduce this burden by simplifying and automating the most tedious and error-prone aspects of the benchmarking process. In this work, we introduce BenchAgent, an AI-powered assistant build on top of the BenchPilot, that allows users to interact with the underlying system in plain language through a user-friendly chat interface, while automatically handling the rest. A central LLM orchestrator interprets user intent and delegates tasks to specialized tool functions, coordinated through a shared memory state that ensures consistency across all agent actions. Through its natural language interface, BenchAgent enables users to query workload-specific documentation, automatically generate YAML configurations through multiple levels of task abstraction, and iteratively refine them through a user-in-the-loop feedback process. Users can further instruct the agent to deploy and monitor experiments in real time, analyze results, and cross-reference them against historical runs — all without directly interacting with the underlying infrastructure. Critically, to accommodate the constrained computational resources characteristic of edge environments, BenchAgent is designed around the principle of task simplification rather than model scaling, progressively abstracting away low-level generation complexity to enable accurate and efficient operation on smaller models.

Keywords: Large Language Models, Agentic Systems, Benchmarking, Edge Continuum

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	Meaning
LLM	Large Language Model
API	Application Programming Interface
RAG	Retrieval-Augmented Generation
AI	Artificial Intelligence
	Abbreviations used in Tables 5.2, 5.3, and 5.1
template-constr-dec	Template-Based YAML Generation with Constrained Decoding
template-free-form	Template-Based YAML Generation with Free-Form Modification
template	Template-Based YAML Generation
free-form	Free-Form YAML Generation

1 Introduction

The rapid proliferation of IoT devices has driven the emergence of Edge Computing as a compelling alternative to centralized cloud processing, enabling data to be analyzed closer to the source and avoiding the latency and bandwidth costs of cloud offloading [1, 2]. A key enabler of this paradigm is the *micro data center* (micro-DC): a small, on-premise cluster of heterogeneous, resource-constrained hardware deployed at the network edge [3]. Evaluating application and infrastructure performance across the large space of configurations and workload conditions that arise in such environments is a significant challenge [4], and one that defines the problem of edge benchmarking.

Edge benchmarking refers to the systematic evaluation of applications and infrastructure components deployed at or near the network edge, where computing resources are constrained and highly heterogeneous. Unlike traditional cloud benchmarking, which operates on relatively uniform, warehouse-scale hardware, edge benchmarking must account for a diverse mix of device types, from low-power ARM-based single-board computers to small multi-core servers, as well as the variability introduced by limited bandwidth, fluctuating network conditions, and factors such as energy consumption and fault occurrence. A meaningful edge benchmark must therefore capture not only application-level performance metrics such as throughput and latency, but also infrastructure-level indicators like resource utilization and power draw. As modern distributed systems increasingly run across such resource-limited and heterogeneous devices, reliable and reproducible benchmarking has become essential for researchers and practitioners alike, yet running such benchmarks remains time-consuming, error-prone, and often requires significant expertise.

Despite its importance, the edge benchmarking landscape remains fragmented. Established suites from the cloud and data management communities were not designed with edge constraints in mind: they commonly assume homogeneous hardware architectures, offer limited support for ARM processors, and require users to manually handle deployment dependencies and system configuration. Several efforts have attempted to address this gap. Plug and Play Bench (PAPB) [5] facilitates big data benchmarking by leveraging containers to run on the underlying execution environment, with a proof-of-concept targeting batch and ML workloads on public cloud providers. Frisbee [6] performs chaos engineering experiments on Kubernetes-deployed containerized applications, offering a declarative language for failure injection and deployment alongside service-level mea-

surement collection. BenchPilot [7] is an open-source, automated framework designed specifically for edge micro-datacenters, aiming to streamline the deployment and evaluation of experiments on edge infrastructures. Despite these advances, a significant barrier to usability persists across the space: configuring experiments, and in particular generating the configuration files (e.g., YAML specifications) that drive these benchmarking execution engines, still requires substantial familiarity with each framework's documentation, workload models, and tunable parameter space. A user wishing to evaluate a new workload or reproduce an existing setup must navigate a non-trivial configuration process before any actual experimentation can begin. The latter introduces friction that slows the research cycle and raises the expertise threshold for effective use, effectively necessitating a benchmarking expert who understands not only how to configure the system, but also comprehends which workloads to deploy under which conditions to obtain meaningful and reliable results.

The rapid advancement of Large Language Models (LLMs) and their demonstrated strengths across a wide range of cognitively demanding tasks, including natural language understanding, multi-step reasoning, structured code generation, and schema-constrained output production, presents a compelling opportunity to address this usability gap [8]. LLMs have shown a remarkable capacity to translate high-level natural language intent into precise, structured outputs, following the specified schema or response format [9–11]. Agentic LLM systems, which extend this capability by equipping models with the ability to invoke external tools, interact with Application Programming Interfaces (APIs), and pursue multi-step goals autonomously, further expand the scope of what can be automated by an intelligent assistant [12–14]. Integrating such a system into the Edge benchmarking pipeline could fundamentally transform the user experience, shifting the interaction from manual, documentation-heavy configuration to intelligent, conversational experimentation that is accessible to users regardless of their familiarity with the underlying framework.

In this work, we introduce BenchAgent, a generative, LLM-driven agentic framework, that automates the full cycle of experimental workflows, from natural language query to result analysis and cross-experiment comparison. BenchAgent is tested on top of the BenchPilot framework [7]. It leverages LLM-based reasoning over workload descriptions, historical experimental results, and system parameters to provide an end-to-end intelligent assistant for edge benchmarking. Rather than replacing the user's judgment, BenchAgent is designed to operate as a responsive and capable collaborator: one that handles the me-

chanical complexity of configuration generation, deployment, and result interpretation, while keeping the human informed, in control, and able to intervene at every stage of the process. Concretely, the system supports the following capabilities:

- Natural-language querying about workloads, past experiments, parameter configurations, stress components, and use cases, enabling users to explore the experimental space conversationally without consulting documentation.
- Automatic generation of BenchPilot-compliant YAML configuration files from user requests, with support for iterative, human-in-the-loop refinement through continued dialogue.
- Execution of generated experiment configurations, with real-time monitoring, log inspection, and failure diagnosis surfaced through the conversational interface.
- Post-experiment analysis through both - structured, statistically grounded evaluation reports that summarize performance metrics, error logs, and system behavior under each tested workload, and LLM-generated summaries based on these reports and raw experimental outputs.
- Comparative evaluation across multiple experiments, leveraging a persistent vector knowledge base to retrieve and cross-reference historical results based on semantic similarity, shared devices, or common workloads.

This integration introduces a new paradigm for interacting with edge benchmarking frameworks, shifting from manual configuration and isolated analysis to intelligent, memory-augmented, conversational experimentation.

1.1 Motivation

The increasing complexity and time overhead associated with configuring and executing benchmarks has become a significant bottleneck in both the benchmarking process and its overall usability. As the number of heterogeneous devices in edge infrastructures grows increasingly , spanning diverse hardware platforms, communication protocols, containerization technologies, and distributed execution frameworks, the space of possible benchmarking configurations expands dramatically. Several tools have been developed [5–7] that provide mechanisms for automating deployment and metric collection. However, they still require users to manually define detailed YAML configurations. These configurations encode not only workload selection and device allocation, but also execution parameters such as duration, resource constraints, replication strategies, and monitoring settings.

In practice, constructing such configurations is far from trivial. Users must invest substantial time in learning the schema, understanding which parameters are relevant, and determining how configurations should be adapted to different hardware setups. Even seemingly simple tasks, such as selecting appropriate benchmarks, choosing meaningful parameter combinations, or writing valid configuration files, can become time-consuming and error-prone. This challenge is further amplified in scenarios where teams are small or lack dedicated benchmarking expertise; for example, when team members are primarily focused on data analysis and are interested mainly in understanding which applications should be deployed on which devices and under what configurations, rather than in the intricacies of benchmarking itself.

Beyond configuration, the analysis of benchmarking results introduces an additional layer of complexity. Interpreting performance metrics, comparing configurations, and deriving actionable insights often requires specialized knowledge. As a result, effectively using benchmarking frameworks frequently demands the presence of a "benchmarking expert", creating a barrier for broader adoption and efficient experimentation.

At the same time, recent advances in LLM-based systems provide a compelling opportunity to address these challenges. Improvements in instruction following, structured output generation, and constrained decoding have enabled LLMs to reliably produce schema-compliant outputs [9–11], which is critical in contexts where incorrect configurations can lead to failed deployments or invalid experimental results. Furthermore, the development of agentic frameworks supporting tool use, multi-step reasoning, and interaction with external systems allows LLMs to participate in end-to-end workflows, including configuration generation, benchmark orchestration, and result processing.

This convergence motivates the central idea of this work: the design of an intelligent benchmarking assistant that automates not only the generation of valid configurations, but also the selection of benchmarks, the exploration of configuration spaces, and the interpretation of results. By effectively acting as a "benchmarking expert", referred to in this work as BenchAgent, such a system aims to significantly reduce the time, effort, and expertise required to conduct rigorous benchmarking, while preserving the reliability, reproducibility, and correctness essential to experimental research.

The overarching aim of this work is to design, implement, and evaluate BenchAgent, an intelligent, agentic LLM-driven assistant that automates the full experimental lifecycle within the BenchPilot edge benchmarking framework, while maintaining a human-in-the-loop interaction model that preserves user oversight and supports iterative, conversational

refinement.

1.2 Contribution

The contributions of this thesis are as follows:

1. **Automated Configuration Generation.** Design and implement a robust pipeline for the automatic generation of BenchPilot-compliant YAML configuration files from natural language user requests. This objective encompasses the investigation of multiple generation strategies — including free-form generation, function-based schema-constrained generation, and constrained decoding — and the evaluation of their relative reliability, token efficiency, and schema compliance rates across models of varying capability.
2. **Agentic Deployment and Monitoring.** Equip the agent with the tools necessary to deploy generated configurations, monitor the execution of experiments, and surface diagnostic information — including service-level and worker-level failure states — through the conversational interface, enabling effective human-in-the-loop oversight of the experimental process.
3. **Automated Result Evaluation.** Develop a structured, statistically grounded evaluation pipeline that processes BenchPilot-generated output files, computes descriptive performance summaries, and produces human-readable evaluation reports that can be interpreted reliably by the language model and stored persistently for future retrieval.
4. **Persistent Knowledge Base and Cross-Experiment Analysis.** Design and implement a vector database-backed knowledge base that stores experimental results and evaluation reports in a semantically retrievable form, enabling the agent to perform cross-experiment comparisons, identify historical trends, and provide contextually grounded responses to analytical queries across sessions.
5. **Memory Management for Conversational Coherence.** Develop a memory management architecture that maintains conversational coherence across multi-turn interactions, manages the context window constraints of the underlying language models, and separates high-level conversational history from low-level task execution traces in a principled and efficient way.
6. **Empirical Evaluation of Design Decisions.** Validate the system's design decisions through empirical evaluation across multiple LLMs of varying scale, assess-

ing schema compliance rates, task completion reliability, retrieval quality, and computational cost, and providing evidence-based guidance on the trade-offs between model capability, generation strategy, and system performance.

1.3 Methodology

The work of this thesis follows a process that progresses from problem formulation to implementation, and finally, evaluation, through three broad phases.

The first phase consists of a literature review and the establishment of the theoretical foundations on which the system is built. This encompasses the study of large language model architectures, decoding strategies, agentic system design patterns, memory management mechanisms, retrieval-augmented generation, and the landscape of existing automated benchmarking frameworks. Particular attention is given to the practical limitations of smaller, locally deployable language models in agentic settings, including their tendency to diverge from instructions or predefined output formats, as well as issues related to context window exhaustion, since these limitations directly influence the architectural decisions made in subsequent phases.

The second phase concerns system design. Drawing on the findings of the literature review, as well as the architecture of BenchPilot, we define the high-level architecture of BenchAgent and establish its core design goals. This phase includes defining the memory management logic, adopting a vector database for persistent experimental knowledge, and designing the tools responsible for interacting with BenchPilot, the local memory space, and the vector database, while maintaining compatibility with resource-constrained deployments.

The third phase consists of empirical evaluation. The system is evaluated across multiple language models of varying scale and capability, including locally-deployed open-weight models from the Qwen, Granite, and Ministral3 families, on a structured evaluation dataset to test different task complexities that BenchAgent is intended to handle. The evaluation encompasses multiple metrics, such as YAML generation accuracy, task completion rates, token efficiency, and latency. The results are examined in the context of the research questions formulated at the outset of the thesis, with particular attention to the trade-offs identified during the design phase. Limitations of the current system are identified and discussed, and directions for future work are proposed that extend the system.

1.4 Thesis Outline

The thesis is organized as follows. Chapter 2 provides the necessary background on large language models, agentic systems, retrieval-augmented generation, and automated benchmarking, establishing the theoretical and technical foundations on which the system is built. Chapter 3 describes the research methodology in detail, covering the supporting libraries and frameworks, and the full implementation of each system component. Chapter 4 presents the research questions, the testbed covering the hardware and model configurations used, the evaluation methodology and the key metrics. Chapter 5 reports and analyzes empirical results across the key dimensions, and discusses the findings. Chapter 6 concludes the thesis and reflects on the limitations and identifies directions for future work.

2 Background and Related Work

2.1 Large Language Models

Large Language Models (LLMs) are a class of deep neural networks trained on vast corpora of text data with the objective of modeling the statistical distribution of natural language [8, 15]. Built predominantly on the Transformer architecture [16], which relies on a self-attention mechanism to capture long-range dependencies between tokens, modern LLMs are characterized by their scale — both in terms of the number of parameters they contain and the volume of data on which they are trained. This scale, combined with training on diverse and broad-coverage corpora, gives rise to a remarkable set of emergent capabilities: the ability to generate fluent and contextually coherent text, to follow complex natural language instructions, to perform multi-step reasoning, and to generalize across tasks without task-specific fine-tuning. These properties have made LLMs the foundation of a rapidly expanding range of applications, from conversational assistants and code generation systems to scientific analysis tools and autonomous agents.

Prominent examples of modern LLMs include GPT-4 [8], LLaMA [17], and Mistral [18], among many others. These models vary considerably in their scale, architecture, training methodology, and intended deployment context — ranging from large, API-accessible proprietary systems optimized for general-purpose capability, to smaller open-weight models designed for efficient local deployment. The distinction between large, highly capable models and their smaller counterparts — sometimes referred to as Small Language Models (SLMs) — is particularly relevant in agentic settings, where the practical constraints of latency, computational cost, and context window size interact directly with the complexity of the tasks being performed [19].

2.1.1 Decoding

At inference time, LLMs generate text autoregressively, meaning that given an input sequence of tokens, the model produces a probability distribution over the vocabulary at each step, and a single token is selected from this distribution to extend the sequence. This process is repeated iteratively, with each newly generated token appended to the input context, until a termination condition is reached — typically the generation of a special end-of-sequence token or the attainment of a maximum length limit. The mechanism by which a token is selected from the predicted distribution at each step is referred to

as the decoding strategy, and it has a substantial influence on the character of the generated output. The simplest strategy, greedy decoding, deterministically selects the single highest-probability token at every step [20]. One of the most common parameters in LLMs regulating the decoding strategy is temperature [21]. Higher temperature values increase the probability of sampling lower-probability tokens, resulting in the selection of less expected words or phrases. This usually leads to more diverse and less deterministic outputs, a feature desirable in creative tasks such as writing. On the other hand, lower temperature values, such as 0, lead to more deterministic outputs and are commonly used in tasks such as mathematical problem solving, where the generated response should remain as consistent and reliable as possible.

2.1.2 Constrained Decoding

While standard decoding strategies operate freely over the full vocabulary at each step, many practical applications require the model's output to conform to a specific structure or schema — such as valid JSON, a well-formed programming language expression, or a domain-specific configuration format. In these settings, unconstrained generation is unreliable: even models with strong instruction-following capabilities can produce outputs that are syntactically malformed, structurally incomplete, or subtly non-compliant with the required schema, particularly under conditions of long output length or complex nesting [9–11]. Constrained decoding addresses this limitation by intervening directly in the token selection process to enforce structural validity at generation time, rather than relying on post-hoc validation or correction [22].

The most principled approach to constrained decoding operates by translating the target output schema into a formal representation, typically a Context-Free Grammar (CFG) or an equivalent finite-state automaton, that defines the complete set of structurally valid token sequences. At each decoding step, the current partial output is matched against this formal representation to determine which tokens in the vocabulary are compatible with at least one valid completion of the sequence. Tokens that would lead to an irrecoverably invalid state are dynamically masked, restricting the LLM from selecting them, ensuring that the model can only select from the subset of tokens that preserve the possibility of a valid final output. This masking operation is applied at every step throughout generation, meaning that structural validity is enforced incrementally and by construction, rather than assessed only upon completion.

This approach has several important practical consequences. First, it provides a hard guar-

antee of output correctness with respect to the specified schema because invalid token sequences are made impossible at the level of the decoding mechanism itself, the resulting output is guaranteed to be structurally valid regardless of the model's underlying tendencies. Second, it eliminates the need for error-prone post-processing steps or retry logic based on validation failures, which are both computationally wasteful and potentially unreliable. Third, it allows smaller and less capable models to reliably produce structured outputs that they might otherwise struggle to generate consistently through instruction-following alone, effectively extending their practical utility to schema-constrained tasks. Libraries such as Outlines [23] and the structured output functionality exposed by the OpenAI API [8] implement variants of this approach.

2.1.3 Quantization and Efficient Inference

The deployment of large language models in resource-constrained environments, such as the edge infrastructure that BenchAgent is designed to support, introduces significant practical challenges related to memory footprint, computational throughput, and energy consumption [24]. A pre-trained LLM with billions of parameters, stored in full 32-bit floating point precision, can require tens to hundreds of gigabytes of memory simply to load, placing it entirely out of reach for local deployment on commodity hardware. Quantization addresses this challenge by reducing the numerical precision of the model's weights, and, in some variants, its activations, from full floating point to lower-bit integer or mixed-precision formats, substantially reducing memory requirements and often improving inference throughput at the cost of a modest reduction in generation quality [25, 26].

The most widely adopted quantization approaches for LLM deployment reduce weights to 8-bit (INT8) or 4-bit (INT4) integer representations [27–29]. Various quantization techniques have been proposed, including Post-Training Quantization (PTQ) [30] and AWQ [25]. More recently, the GGUF format popularized by the llama.cpp [31] inference library, which is employed in this work for local model deployment, has enabled flexible, mixed-precision quantization that allows individual layers or weight matrices to be quantized at different bit-widths, supporting fine-grained trade-offs between model size and generation quality on consumer-grade hardware.

Quantization is particularly consequential in the agentic setting because it affects the fluency of the model's natural language generation and its instruction-following reliability [26, 32]. Highly quantized models may experience reduced reliability in generating syn-

tactically complex structured outputs, such as nested JSON or YAML, especially at low bit-widths, as quantization introduces approximation errors in model parameters that can impact the stability of token-level predictions and long-range structural consistency.

2.1.4 Context Window

The context window of a language model defines the maximum number of tokens that can be processed in a single forward pass, encompassing the full input, including the system prompt, conversation history, tool descriptions, retrieved documents, and any intermediate reasoning or tool outputs. This limit is a fundamental architectural constraint that has profound implications for agentic system design, where the effective context consumed by a single session can grow rapidly and unpredictably.

Modern LLMs support context windows ranging from tens of thousands to over a million tokens in the most capable API-accessible models. However, despite these advances, long contexts introduce their own challenges beyond raw capacity. Research has shown that language models exhibit a systematic tendency to attend more strongly to information at the beginning and end of the context window, with content in the middle of a long context receiving systematically less attention, a phenomenon referred to as the "lost in the middle" effect [33, 34]. This has direct implications for the design of agentic systems that inject retrieved documents or tool outputs into the middle of a long context, and motivates careful attention to the ordering and placement of information within the prompt. Furthermore, for smaller and locally-deployed models, practical context window limits remain substantially lower than those of frontier API models, requiring active context management strategies, including summarization [35,36], selective retention [37,38], and the external memory mechanisms [39] described in subsequent sections, to maintain coherent multi-turn operation within the available token budget.

2.1.5 Small Language Models and On-Device Deployment

While the majority of public attention in the LLM literature has focused on the largest and most capable models, a parallel and increasingly important research direction concerns the development and deployment of smaller, more efficient models that can be run locally on commodity or resource-constrained hardware. These models — variously referred to as Small Language Models (SLMs), efficient LLMs, or edge-deployable models — occupy the portion of the capability-efficiency frontier where inference can be performed without access to high-end GPU clusters or cloud API services, making them particularly relevant

to the edge computing context that motivates this work, and extends the prior work on the SLMs in agentic systems [40–42].

The development of SLMs has been driven by several complementary techniques. Knowledge distillation [43, 44] trains a smaller student model to mimic the output distribution of a larger teacher model, transferring capability without transferring scale.

In the context of agentic systems, SLMs present both opportunities and challenges. On the opportunity side, local deployment eliminates API latency, removes dependency on external service availability, avoids per-token billing costs, and addresses data privacy concerns that may preclude the use of cloud-hosted models in sensitive research contexts. On the challenge side, SLMs typically exhibit weaker instruction-following behavior, and greater susceptibility to reasoning drift than their larger counterparts, all of which are particularly consequential in agentic settings where errors compound across steps.

2.1.6 Prompt Engineering

The behavior of a large language model at inference time is determined not only by its weights, shaped during pre-training and fine-tuning, but also by the structure and content of the input it receives. Prompt engineering is the discipline concerned with the deliberate design of model inputs to elicit outputs that are more accurate, reliable, consistent, or appropriately formatted for a given task [45]. While early work in this area focused primarily on the phrasing of individual queries, the field has matured considerably alongside the models themselves, encompassing a rich set of techniques that operate at the level of instruction design, contextual scaffolding, and the strategic organization of information within the input sequence.

Among the most influential and widely adopted prompting techniques is chain-of-thought (CoT) prompting [46], which encourages the model to produce explicit intermediate reasoning steps before arriving at a final answer. By externalizing the reasoning process into the generated text, CoT prompting has been shown to substantially improve performance on tasks that require multi-step inference, arithmetic, or logical deduction, tasks on which direct answer generation tends to fail. Few-shot prompting [47], introduced alongside GPT-3, provides the model with a small number of input-output examples directly within the prompt, allowing it to infer the desired task format and behavior. In the zero-shot setting, where no examples are provided, the careful formulation of the instruction itself becomes the primary lever for shaping model behavior. In agentic systems, prompt engineering extends beyond individual query design to encompass the construction of system

prompts that define the agent's persona, capabilities, and behavioral constraints; the formulation of tool descriptions that communicate intended use clearly enough to support reliable invocation; and the design of error recovery instructions that guide the model toward productive responses when tool calls fail or produce unexpected outputs. The quality of these prompting decisions has a direct impact on the reliability of agentic task completion, and is therefore taken into account in the design of BenchAgent.

2.2 Agentic Systems

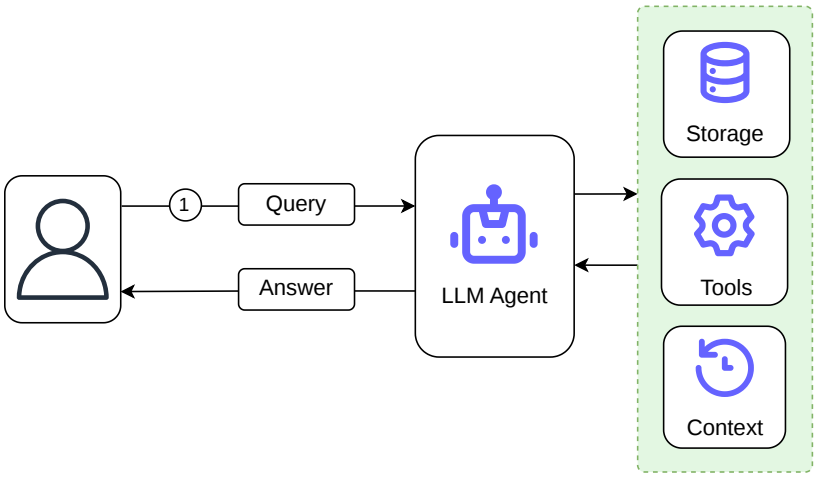

Figure 2.1: Components of an Interactive Agentic System

Agentic LLM systems represent a significant evolution beyond the standard single-turn or conversational use of language models. Rather than responding to isolated queries, an agentic system places the LLM at the center of a broader decision-making loop in which the model is tasked with pursuing goals over multiple steps, selecting and invoking external tools, interpreting the results of those invocations, and iteratively refining its actions based on feedback from the environment. This shift substantially expands the range of tasks that LLM-based systems can address, enabling them to interact with APIs, execute code, query databases, browse the web, manage files, and coordinate complex multi-step workflows that would be impossible to complete within a single generation pass. The model is no longer a passive responder, but an active agent, operating either autonomously or with a human-in-the-loop inclusion, which allows the user to validate intermediate outputs, provide clarification, or approve consequential actions before they are executed — and is particularly valuable in domains where errors are costly or difficult to reverse.

The typical agentic architecture follows a cyclical reasoning and action loop, most influentially formalized in the ReAct framework [48], which interleaves natural language reasoning steps with concrete actions and their observed outcomes. At each iteration of the loop, the model executes an action through a tool or API call, observes the result, and incorporates the observation into its context. This loop continues until the model determines that the task has been completed or that further progress is not possible. More recent frameworks, such as smolagents [49], LangChain [50], and LlamaIndex [51], have operationalized this paradigm into practical development libraries that provide scaffolding for tool registration, prompt management, execution loop control, and multi-agent coordination.

2.2.1 Agentic Memory Mechanisms

A defining characteristic of agentic systems is their need to maintain and manage information across multiple steps and, in conversational chat-bots, across multiple turns of interaction. This requirement is fundamentally at odds with the stateless nature of individual LLM inference calls, which have no intrinsic memory of prior inputs beyond what is explicitly included in the current context, and becomes increasingly critical as agents are deployed in long-horizon, dynamic environments where sustained reasoning and task execution place persistent demands on available context capacity [52, 53].

Short-term memory, also known as "working memory," in an agentic system refers to the information maintained within a single or, in the case of interactive systems, multiple agentic loops forming an active dialogue [54, 55]. It typically stores selectively retained information from recent interactions, including previous user inputs, final agent responses, and, in some cases, intermediate tool outputs or error traces considered useful for future behavior correction. The selection process is usually heuristic or retrieval-based, ranging from simple recency-based retention, such as keeping the last n interactions, to more structured filtering strategies that prioritize informative or failure-inducing episodes. This form of memory enables continuity within an active dialogue session, allowing the agent to reuse prior experience, avoid repeating past mistakes, and improve response consistency. However, short-term memory remains bound to a single interaction session between the user and the agent and does not persist across separate sessions.

Long-term memory, on the other hand, refers to information stored externally, such as in databases, allowing it to persist across multiple interaction sessions [55]. Recent approaches have incorporated graph databases and vector-based retrieval mechanisms to ex-

tend memory capacity and improve retrieval capabilities. Several works have focused on efficient agentic memory systems [56, 57], which, however, are unable to establish novel connections or adapt their organizational structures as knowledge evolves. To solve this, another line of work has focused on incorporating dynamic memory refactoring [58]. These approaches highlight the growing importance of adaptive and scalable memory architectures for enabling more effective long-term reasoning and knowledge management in agentic systems.

2.2.2 Tool-calling

One of the most consequential components of agentic systems is the concept of tool calling — the mechanism by which an LLM delegates specific operations to external, deterministically defined functions. Tools are user-defined or framework-provided functions that enable the model to perform operations that lie outside the scope of pure language generation: querying databases, executing shell commands, calling external APIs, performing arithmetic, managing files, or interacting with any other programmatic interface. By equipping the model with a curated set of tools, the system designer effectively defines the action space available to the agent — the set of interventions it can make in the external environment in pursuit of its assigned goals.

During tool execution, the LLM itself remains idle: having produced a structured tool call specifying the function to be invoked and the arguments to be passed, the function is invoked deterministically. This separation between generative and deterministic computation is architecturally important: it ensures that operations with well-defined correct answers, such as database lookups or file reads, are handled deterministically rather than with approximation, reserving the model's generative capacity for tasks that genuinely require natural language reasoning. The result of the tool call is then injected back into the model's context as an observation, allowing the model to interpret the output and plan its next action accordingly.

For a tool to be usable within an agentic framework, it must be specified using a structured schema that exposes sufficient information for the model to invoke it correctly. This schema typically includes the tool's name, a natural language description of its functionality and intended use, the names and types of its input parameters along with semantic descriptions of each, and a description of the expected output format. This is used by the framework to validate tool calls at execution time, and it is included in the model's prompt to inform its understanding of what each tool does and when it should be invoked. The

quality and clarity of tool descriptions therefore has a direct and significant impact on the reliability of tool use, poorly described tools are more likely to be invoked incorrectly, with malformed arguments or in inappropriate contexts, leading to execution failures that consume steps, inflate token counts, and degrade overall task completion rates.

2.2.3 Vector Databases

The storage and retrieval of information in agentic and RAG-based systems introduces requirements that are fundamentally misaligned with the design assumptions of traditional database systems. Relational databases, optimized for structured, exact-match or range-based queries, are ill-suited to the retrieval of semantically similar content from collections of unstructured natural language documents. Vector databases were developed to address precisely this gap, providing a storage and retrieval infrastructure purpose-built for high-dimensional embedding vectors and the approximate nearest-neighbor queries that semantic search requires.

A vector database stores each document as a dense numerical vector — its embedding — produced by an encoder model such as a sentence transformer, alongside any associated metadata and the original text content [59]. At query time, the query is encoded into the same embedding space using the same encoder, and the database retrieves the stored vectors whose geometric proximity to the query vector, as measured by a distance or similarity metric such as cosine similarity or Euclidean distance, exceeds a relevance threshold or falls within the top-k results. Because exhaustive comparison of a query vector against every stored vector becomes computationally prohibitive at scale, vector databases employ approximate nearest-neighbor (ANN) indexing structures — most prominently Hierarchical Navigable Small World graphs (HNSW) [60] and Inverted File Indexes (IVF) [61] — that trade a small, controllable degree of retrieval exactness for substantial gains in query throughput and scalability. HNSW in particular has become the dominant indexing structure in production vector database systems due to its favorable balance of query speed, recall quality, and incremental update support. Prominent vector database systems include Pinecone [62], Qdrant [63], Milvus [64], and ChromaDB [65] — the latter of which is employed in this work due to its lightweight architecture that requires no external server infrastructure and integrates naturally into a locally-deployed agentic system. Beyond pure vector search, modern vector databases support hybrid retrieval strategies that combine dense embedding-based retrieval with lexical filtering over metadata fields. For example, a stored embedding may contain a metadata field such as “filename,” indicating

the file from which the embedding originated. This metadata can then be leveraged during querying to restrict the search within a specific file, enabling more targeted retrieval. This enables queries that simultaneously exploit semantic similarity and structured attribute constraints — a capability that is directly leveraged in BenchAgent's retrieval logic to support both content-based and field-constrained access to the experimental knowledge base.

2.2.4 Retrieval-Augmented Generation

A fundamental limitation of LLMs is the finite context window, as reasoning over large corpora, or locating relevant chunks within a broader document set, quickly exceeds what can be held in a single prompt. Retrieval-Augmented Generation (RAG) directly addresses this by equipping the model with a mechanism to fetch precisely the context it needs at inference time, enabling more accurate and context-aware responses without requiring the entire knowledge base to be present upfront [66–68]. Building on this foundation, graph-based approaches such as GraphRAG [69] further enrich retrieval by introducing structured representations, entity graphs and community summaries, that capture complex interdependencies within the data and enable question answering over highly interconnected knowledge. While graph-based RAG captures richer semantic structure across document collections, it typically requires heavy preprocessing, manual schema design, and extremely costly updates of the data, since the community summaries and the relationships need to be reistablished, making it difficult to use in dynamic environments [70]. This led to a line of research, such as LightRAG [70], which further combine hierarchical or dual-level retrieval with incremental updates, improving both comprehensiveness and efficiency for large, dynamic text collections. Recent work proposes hierarchical-based RAG systems, leveraging multiple agents for efficient routing to vector, graph, and web-based databases based on the scope of the question [71].

2.3 Automated Benchmarking

While benchmarking is essential for evaluating distributed systems, configuring and executing benchmarks is often tedious and time-consuming. This challenge is amplified in heterogeneous edge and micro-DC environments, where the large space of hardware configurations, workloads, and system parameters makes exhaustive manual evaluation impractical. Standard benchmarks also rarely capture user-specific requirements, and meaningful configuration demands deep domain expertise. Recent work has explored

leveraging LLMs to automate aspects of this process, such as applying pre-trained language models to analyze manuals and textual guidance [72], and using LLMs to transform user prompts into executable benchmarking plans [73]. These approaches demonstrate the potential of LLMs and agentic systems for automating configuration and benchmarking tasks. Building on these insights, we extend these ideas to edge and micro-datacenter environments.

2.3.1 BenchPilot

BenchPilot [7] is an open-source, auto-benchmarking framework designed specifically for edge micro-DC environments. It addresses a well-known gap in the benchmarking landscape: while established benchmarks from the data management and cloud computing communities typically provide workloads and data generators, they place the burden of dependency management, hardware and software configuration, and infrastructure compatibility squarely on the end user. This burden is particularly prominent at the edge, where heterogeneous hardware architectures — such as ARM-based processors — require executables to be rebuilt and configurations to be carefully adapted to the underlying infrastructure. Moreover, existing tools tend to focus exclusively on application-level performance metrics, overlooking infrastructure-oriented indicators such as energy consumption, and offer limited extensibility in their metric collection pipelines. BenchPilot tackles these challenges by leveraging cloud-native technologies — infrastructure-as-code and containerization — to abstract hardware heterogeneity and automate the full evaluation lifecycle.

The entry point for any BenchPilot experiment is a user-defined YAML configuration file. The user specifies the target infrastructure, the chosen workload, and its parameters — such as the processing engine, the number of nodes, parallelism settings, and experiment duration. Once the YAML is defined, BenchPilot takes full ownership of the execution - it provisions the environment, deploys the workload, orchestrates the experiment, and tears down the deployment upon completion. The framework ships with a set of ready-to-use, containerized benchmarking workloads that users can select and deploy out of the box, while remaining fully extensible to support custom workloads.

Upon completion, BenchPilot persists the collected measurements in structured `CSV` and `JSON` files, capturing both application-level and infrastructure-level metrics. Application metrics include throughput and end-to-end latency, while infrastructure metrics cover CPU and memory utilization, network traffic, and energy consumption across the nodes

of the micro-DC. This dual focus on application and infrastructure observability — across heterogeneous, resource-constrained hardware — is what distinguishes BenchPilot from conventional benchmarking tools.

However, the complexity of authoring and iteratively refining the YAML configurations, deploying, as well as performing the post-experiment analysis of the results, still persist as bottlenecks to inexperienced users, since it requires knowledge of the available workloads, their configurable parameters, and the underlying infrastructure. These are precisely the problems that BenchAgent is designed to solve.

3 BenchAgent Framework

BenchAgent is packaged and deployed as a Docker container, hosted on the same local server as the BenchPilot framework. This co-location grants the backend — which encompasses the orchestrating agent, all tool implementations, and the session memory — direct access to the internal execution commands of the BenchPilot, such as issuing deployment commands, invoking troubleshooting routines, and reading experiment result directories and output files. Placing this logic remotely would necessitate additional communication layers between the agent and the framework, reintroducing the kind of complexity and potential failure points that the overall design seeks to minimize. All backend components are implemented in Python and reside within the same container, sharing a unified runtime environment.

The user interface is implemented using the *Streamlit* [74] Python library and is served from within the same container. Since all relevant state and agentic logic are managed by the backend, the UI is responsible solely for rendering the conversation and forwarding user input to the orchestrating agent. It serves as the primary medium through which requests are issued, feedback is communicated, and results are surfaced to the user.

3.1 Core Framework Components

BenchAgent's architecture is guided by the principle of minimizing agentic complexity — reducing LLM calls, narrowing each decision point, and delegating as much logic as possible to deterministic components. In practice, this means consolidating related operations into high-level tools rather than exposing many fine-grained ones, which reduces latency, limits error propagation, and decreases reliance on the model for low-level sequencing. Equally important is information flow: since the LLM operates solely on its current context, all relevant state must be communicated explicitly through careful prompt design and informative tool outputs.

Below we describe the core generic components of BenchAgent — those that apply across all tools and tasks — along with the libraries and frameworks that underpin them.

3.1.1 Agent Orchestration

To manage the interaction between the LLM and external tools, we employ SmolAgents [49], a lightweight framework that simplifies the development and deployment of agentic

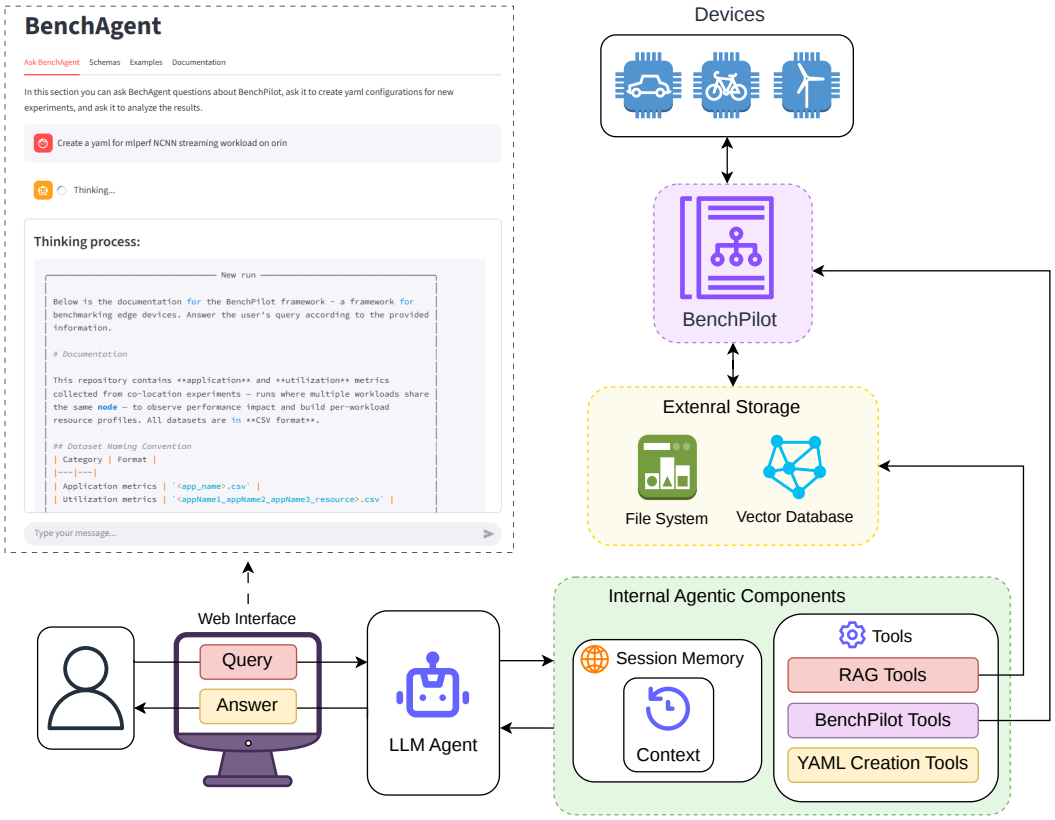

Figure 3.1: Components of BenchAgent

systems. SmolAgents provides a structured interface for orchestrating tool calls through its ToolCallingAgent, where tool invocations are generated as JSON objects. This design aligns with widely used API standards and is model-agnostic, supporting multiple backends and API formats.

To standardize interactions across different model backends, all requests are routed through the OpenAI API as a unified communication layer. Rather than directly invoking model-specific endpoints, this abstraction decouples the agentic framework from backend-specific implementation details, enabling different models to be swapped or evaluated under identical conditions without modifying the surrounding infrastructure.

3.1.2 Memory Management

Memory management is among the most architecturally crucial components of an agentic system, and its importance is amplified significantly in systems that are designed around

human-in-the-loop or conversational interaction paradigms. Unlike stateless query-response systems, a conversational agent must maintain coherent awareness not only of the current user's query, but also of their previous interactions and the produced responses. Without principled memory management, the agent loses the ability to interpret references to prior context, build incrementally on earlier results, or maintain consistency across the turns of a multi-step investigative session. Therefore, a mechanism is required to manage the short-term, intra-session memory that governs the agent's moment-to-moment coherence within a single conversation.

This challenge is further complicated by the finite context window imposed by all current language models — a constraint that becomes particularly acute in agentic settings, where the raw context consumed by a single session can grow rapidly. In addition to the user's messages and the agent's responses, the context must accommodate the agent's intermediate reasoning steps, the inputs and outputs of every tool invocation, framework-level system prompts, and any retrieved documents injected via RAG. For large, capable models this constraint is manageable, though never trivial; for smaller language models operating under tighter token budgets, it becomes a primary architectural concern. Practical implementations must therefore incorporate active information management strategies — such as selective retention, context summarization, and bounded history depth — to prevent the context window from being exhausted and to maintain generation quality as the interaction deepens.

We define two main memory components that together form the agent's available context and knowledge. The first is the *Session Memory*, a temporary, session-scoped memory that stores the context and variables shared between the LLM and its tools; it exists only for the duration of a single user interaction. The second is the *Persistent Memory*, a more detailed, long-term storage layer that records fine-grained information such as interaction duration, individual tool invocation data, and response times (e.g., ask-to-answer latency). Unlike the Session Memory, this memory is stored on disk and remains accessible after the session ends. Conceptually, the Session Memory operates as a transient subset or working view within the broader Persistent Memory. Next, we describe a context retrieval function that serves as a reference point for cases where the model has lost track of the original user query amid a long sequence of tool invocations, whether due to context window truncation or simply the complexity of the agentic execution.

3.1.2.1 Session Memory

We implement a *Session Memory* — a class that stores and exposes the information necessary to construct the appropriate context for each tool invocation. Our architecture employs a single main LLM agent as the central orchestrator, responsible for interpreting user requests and invoking the appropriate tools. Crucially, this orchestrator has no awareness of the internal implementation of any tool, nor of any LLM invocations that may occur within the tools. Despite this separation, all tools and LLM instances share access to the same session memory, which serves as the *sole medium of coordination* across the system. It is composed of four key variables.

The first is the *current user query*, which captures the user's most recent request. Since it is stored in a variable, it can be explicitly used as a parameter by different tools, appended to prompts, or used to perform vector search in our vector database, whenever needed.

The second is the *current YAML configuration*. This variable is important for the task of YAML generation, which is described in more detail in section 3.2.1. Once generated and successfully verified, the YAML is stored in this variable, where it persists across the session and can be reused and progressively refined by subsequent tasks, such as modifying experiment parameters or issuing deployment commands. The *current YAML configuration* is not freely mutable. While multiple tools may read from and write to it, only successfully verified results are allowed to update the YAML. This ensures the shared context remains consistent and trustworthy throughout the session, while also eliminating the risk of overwriting a previously successful YAML generation with an invalid YAML.

The third variable is the *current conversation context*, which maintains session-level information. It represents the high-level conversational record of the session — the sequence of user queries and agent responses as they would appear in a clean, human-readable dialogue. Because this history grows with every exchange, earlier user-question - agent-answer pairs, are pruned to prevent the accumulated dialogue from consuming the available context budget. This variable provides fine-grained control over what context is passed to the LLM at each invocation, enabling the system to selectively transform, filter, and adapt the conversational history as needed.

The fourth variable is the *current exchange context*, which stores the most recent query–response pair along with selected information such as tool invocation failures and errors. This context is passed back to the agent to prevent repetition of the same mistakes. It is discussed in more detial in section 3.1.3.

3.1.2.2 Persistent Memory

The *Persistent Memory*, records the full internal trajectory of the agent's activity in response to each user query, including every tool invocation issued, the inputs provided to each tool, the responses returned, the number of reasoning steps taken, and any intermediate outputs or error states encountered along the way. This fine-grained record is essential for transparency, debugging, and the kind of targeted human-in-the-loop intervention described in earlier sections, where the user may wish to understand not just what the agent concluded, but how it arrived at that conclusion. The persistent memory is richer and more verbose than the session memory, and is therefore managed separately to avoid polluting the conversational context with low-level execution details that are rarely relevant to the user's immediate informational needs. The relationship between the two components is as follows: when the agent produces a final response for the user's query, this final response — as extracted from the persistent memory — is used to update the session memory, appending the agent's reply to the conversational record. This separation ensures that the session memory remains a clean, high-signal representation of the dialogue, while the persistent memory serves as a complete and auditable log of the agent's internal process.

3.1.2.3 Context Retrieval

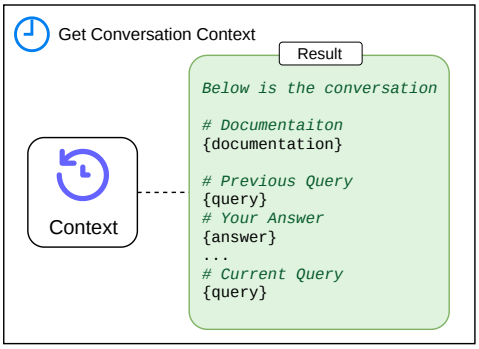

Figure 3.2: Conversation Context Retrieval Function

One approach to handling the limited context window was to allow the model to retrieve the conversation context at any point during execution. The implementation of this function can be seen in Figure 3.2. The motivation behind this is that When a model makes multiple unsuccessful tool invocations in sequence, the initial conversation context may have been truncated due to context window constraints. This is a particularly serious issue

in agentic systems, where the number of tokens tends to grow rapidly as the agent reasons, invokes tools, and processes their results across many steps. By providing the model with explicit access to the full conversation context, it can re-orient itself and continue attempting to resolve the task, rather than proceeding on a truncated or incomplete understanding of what was originally asked.

3.1.3 Error Handling and Recovery

Error handling is a central concern in agentic systems, where failures at any step — whether due to schema violations, tool misuse, invalid file references, or model-side generation errors — can propagate through the pipeline if not properly contained. In standard software systems, errors are typically caught at well-defined boundaries and handled recovery procedures. However, in agentic systems, the recovery mechanism is itself a language model, which means that the quality of error recovery depends directly on the quality of the information the model receives about what went wrong. A silent failure or an ambiguous error message leaves the model without the context it needs to correct its behavior, which may result in repeated invocations of the same failing tool with identical or marginally modified arguments, leading to wasted tokens, inflated latency, and an unresolved underlying issue.

To address this, every BenchAgent tool is designed to return explicit descriptive error messages upon failure, providing sufficient context for the model to diagnose the problem and select an appropriate corrective action on its next invocation. These messages are automatically appended to the agent's observation history, ensuring that the model has direct and unambiguous visibility into the nature of the failure before it next produces output.

The session memory presented to the model across question-answer turns includes only the user's query and the final angent's answer, without exposing the intermediate tool invocations and error messages. This is a deliberate design choice that reflects the tension between context efficiency and recovery capability: surfacing the full execution trace of every prior turn would rapidly exhaust the available context budget, particularly for smaller models, while providing limited benefit given that most inter-turn context can be adequately conveyed through the high-level conversational record. However, within a single query session, full failure visibility is preserved — all errors encountered during the current exchange are visible to the model in their entirety, enabling effective within-turn recovery without imposing a cross-turn context cost.

3.1.4 Prompt Engineering Strategy

Effective prompt engineering is essential in a system where the agent must reliably orchestrate a structured, multi-step workflow across models that range from large, highly capable instruction-tuned systems to small, resource-constrained ones operating under tight token budgets. A prompting strategy that is well-calibrated for a frontier model may be entirely inadequate for a smaller one, and vise versa — yet the prompting infrastructure must be designed to support both, since model selection is itself a variable in the system's evaluation. BenchAgent's prompting strategy is accordingly built around two core principles that apply universally across model scales: *completeness* and *conciseness*.

Completeness requires that the model always has access to the full set of information necessary to complete its current task, regardless of how many prior exchanges have occurred in the session. In practice, we re-supply the full system documentation at every invocation. This is necessary because BenchAgent truncates the interaction history to a bounded window of recent exchanges in order to manage context window constraints — a consequence of which is that instructions provided only at the start of a session may no longer be present in the model's context by the time a later query is processed. Without re-supplying this information, the model risks operating on an incomplete view of its constraints and producing outputs that deviate from the expected structure or ignore critical behavioral requirements.

Conciseness, the complementary principle, requires that the prompt contain only information that directly contributes to the task at hand. This is particularly consequential when targeting smaller models, where long prompts consume a disproportionate share of the available context window, leaving less capacity for the model's reasoning and output, and where excessive context has been shown to degrade instruction-following quality [75].

Together, these two principles define a prompting posture that is simultaneously maximally informative and minimally wasteful — providing the model with everything it needs and nothing it does not.

3.2 BenchAgent Functionalities

3.2.1 YAML Configuration Generation

The creation of YAML configurations is a critical part of the BenchPilot system. When performed manually, it demands a deep understanding of the supported workloads, their

parameters, and their allowed or recommended values, as well as familiarity with BenchPilot's documentation. To lower this barrier, one of the key capabilities of our system is the ability to issue natural language queries to BenchAgent, which automatically generates the necessary YAML configuration tailored to the user's requirements.

We evaluate multiple approaches for generating YAML configuration. First, we need to account multiple factors, which include the LLM's context window, the LLM's size, and the LLM's capabilities. Crucially, as edge continuum is characterized by their limited computational resources, they can either bottleneck the deployment of larger, more capable models or render such deployments infeasible altogether, thereby challenging efficient and accurate YAML generation. To address this, the model should ideally perform as little computation as possible while focusing on higher-level abstractions, thereby accommodating its limited capabilities. For this purpose, we evaluated multiple approaches operating at different levels of task abstraction, which are explained in the following chapters: (a) free-form YAML generation, (b) template-based YAML generation with free-form modification, (c) template-based YAML generation, and (d) template-based YAML generation with constrained decoding. Across these approaches, we systematically reduce the model's direct interaction with the raw YAML configuration by introducing intermediate abstractions that act as guardrails, steering the model away from error-prone, low-level generation.

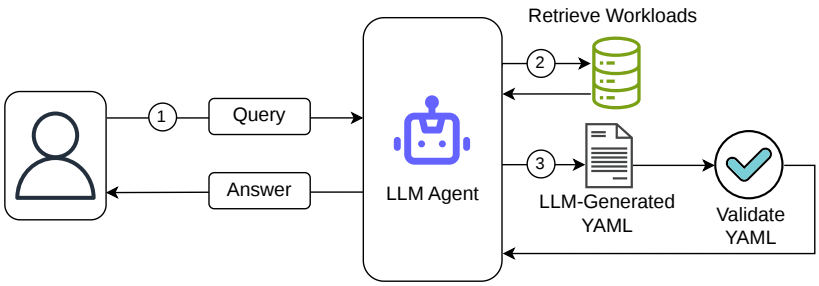

Figure 3.3: Free-Form YAML Generation

3.2.1.1 Free-form YAML

In this setting, we allow the model to generate YAML configurations in a free-form manner, meaning that the output is produced without any constraints during generation. Instead, the approach relies entirely on the model's ability to produce a valid configuration from the provided examples. For this, we provide two key tools that guide and validate its

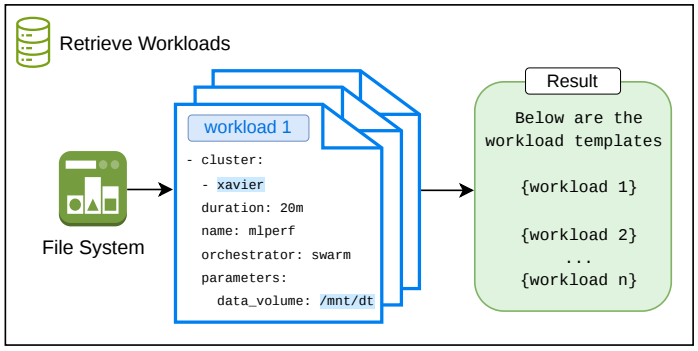

Figure 3.4: Retrieves all the available workloads

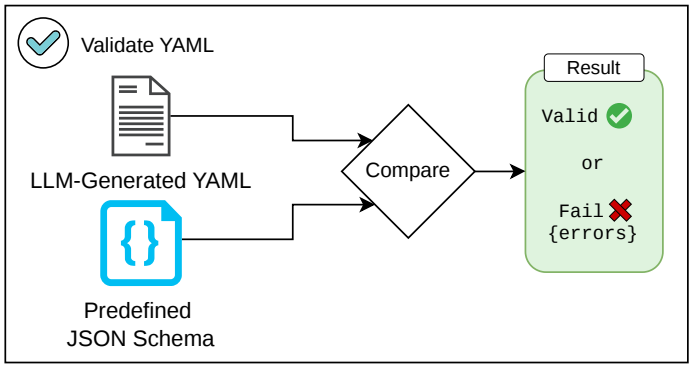

Figure 3.5: YAML Validation

outputs. The Figure 3.3 illustrates this YAML generation pipeline. First, the model is provided access to a schema retrieval function, shown in Figure 3.4, which returns a collection of example workload configurations. These examples define the expected structure, required fields with their default values, and, if needed, with other permissible values and their types, for different workloads. These workloads are extracted from the predefined schema that we produce in 3.2.1.1.1. By calling this tool, the model can ground its generation process in templates that are compatible with the BenchPilot framework. Second, we introduce a verification function that evaluates the syntactic and structural correctness of the generated YAML, shown in Figure 3.5. The model submits its generated configuration as input to this function, which validates it against the predefined generic schema. If the YAML is valid, the process saves the YAML in the *current YAML configuration* variable of the session memory, and terminates successfully. Otherwise, the verifier returns detailed error messages describing the violations, such as missing fields, incorrect data

types, or invalid nesting. In this way the agent can iteratively refine the generated YAML based on the verifier feedback, and resubmit it for validation, repeating this process until a valid configuration is produced, or until the maximum number of steps in the agentic loop is reached.

3.2.1.1.1 Predefined Schema Generation A prerequisite for validating LLM-generated YAML configurations is the existence of a reference schema that formally specifies the structure that a valid configuration must conform to. Maintaining such a schema manually is both labor-intensive and fragile: as BenchPilot evolves and new workload types or configuration parameters are introduced, a hand-authored schema must be updated in lockstep with the framework, creating an ongoing maintenance burden and a persistent risk of divergence between the schema and the actual valid configuration space. To address this, we adopt an automated schema derivation approach that eliminates the need for manual schema authoring entirely.

Rather than specifying the expected configuration structure by hand, we automatically derive a predefined JSON schema from a collection of previously validated, real-world BenchPilot YAML configurations. This is accomplished using the `SchemaBuilder` provided by the Genson [76] python library, which analyzes the structural properties of a set of input documents and constructs a schema that captures the union of all structural patterns observed across them — including all encountered field names, their inferred types, and their nesting relationships. The resulting schema is therefore grounded directly in the space of configurations that BenchPilot has previously accepted and executed successfully, providing a reference structure that is both empirically validated and automatically kept in sync with the evolving configuration space as new examples are collected. This schema acts as the formal ground truth against which LLM-generated configurations are validated before deployment, catching structural errors before they can cause runtime failures, and it is additionally exposed through the user interface to allow manual inspection and modification when domain-specific constraints or custom configurations require human oversight. This approach significantly lowers the engineering burden associated with schema maintenance while ensuring that the validation reference remains firmly rooted in real, previously validated experimental practice.

3.2.1.2 Template-based YAML Generation with Free-form Modification

To ensure strict adherence to the target configuration schema, we adopt a function-based generation strategy that constrains the model's outputs through statically defined input formats. These formats are implemented using the Pydantic python library [77], which enables validation of structured inputs before any generation or modification is performed. Rather than generating YAML configurations in a single free-form step, the process is decomposed into two sequential stages: (i) template construction and (ii) parameter modification. This decomposition separates structural correctness from content customization, in this way breaking down the task into more manageable steps. The pipeline can be seen if Figure 3.6.

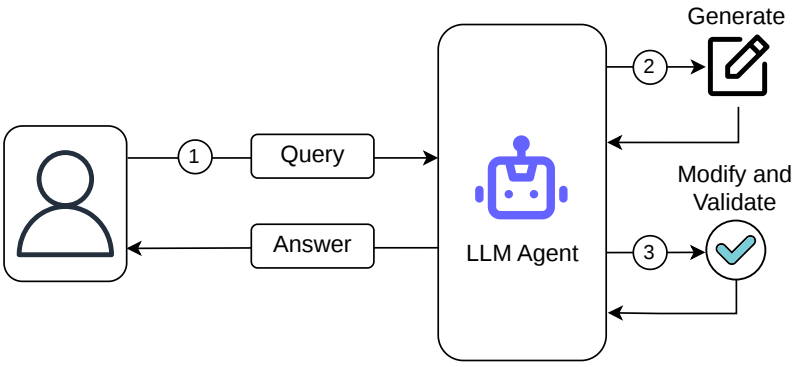

Figure 3.6: Template-Based YAML generation with Free-form Modification

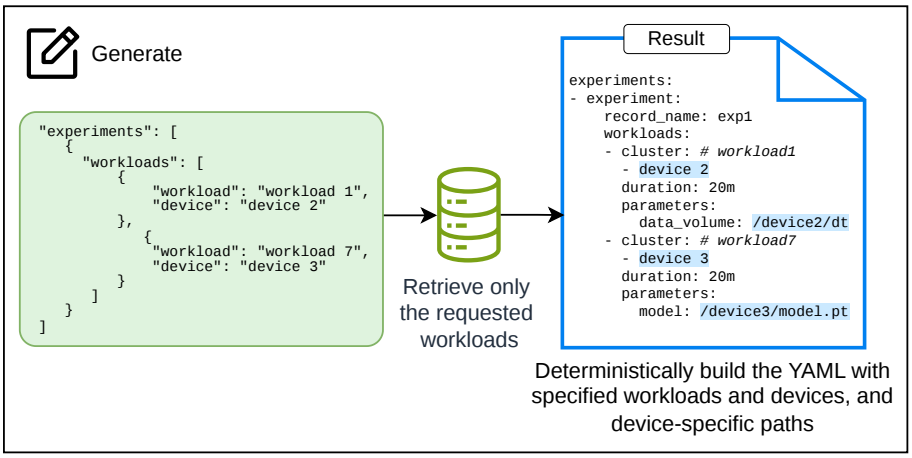

Figure 3.7: Template-Based YAML Generation Function

3.2.1.2.1 Template Construction. In the first stage, the model invokes a template-building function responsible for generating a valid YAML configuration populated with default values. The function enforces a static input schema, defined via Pydantic, which specifies only the high-level components of the experiment — namely, the selected workloads and their corresponding devices, as can be seen in Figure 3.7. The required input structure is explicitly provided in the tool description. If the model produces an input that does not conform to this schema, validation fails, and the agent can retry the function call. Once valid input is received, the function deterministically constructs the YAML configuration by:

- Retrieving predefined workload templates based on what it received in the input
- Injecting device-specific information (e.g., paths, RAM, threads)
- Replacing placeholder variables (e.g., device identifiers, workload paths)
- Assembling the final configuration

Additionally, the function automatically generates a unique record identifier for each configuration by checking for naming collisions within the target directory and appending suffixes where necessary, ensuring that new experiments do not overwrite existing YAML files. The produced YAML is then saved in the *current YAML configuration* variable of the session memory.

This keyword-based template generation approach guarantees that the produced YAML is structurally valid by construction, regardless of the complexity or nesting depth of the configuration. Critically, it removes the need to present the model with full schema examples, which would substantially inflate the prompt token count and risk complicating the task. In a free-form generation setting, producing such structures token-by-token substantially amplifies the probability of diverging from the correct schema, as even minor deviations in indentation, field naming, or value typing can render the entire configuration invalid. By instead delegating structural assembly to a deterministic function, the model is relieved of this burden entirely — correctness becomes a property of the construction mechanism rather than a consequence of the model's generation accuracy.

3.2.1.2.2 Parameter Modification. With a structurally valid template in hand, the second stage handles content customization. Here, the model invokes the validation function 3.5 — described to the model as a modification function to guide it toward using it exclusively for modification purposes — where the model must provide as input the full YAML

configuration with the changed parameters. The motivation is that the model now operates over a considerably smaller YAML configuration, rather than over all the possible workloads, which it can manipulate more easily and with fewer decisions to make.

This two-stage design yields key practical benefits that directly follow from this separation of concerns. First, it drastically reduces the number of tokens the model must generate at each stage: template construction requires only a compact, high-level specification of workloads and devices, while parameter modification operates over an already-structured configuration rather than producing YAML from scratch. Secondly, this approach eliminates the need to present the model with verbose workload schema examples in the prompt, keeping the prompt compact, focused, and unambiguous, and ultimately reducing the risk of the model misinterpreting the task or producing invalid outputs.

3.2.1.3 Template-based YAML generation

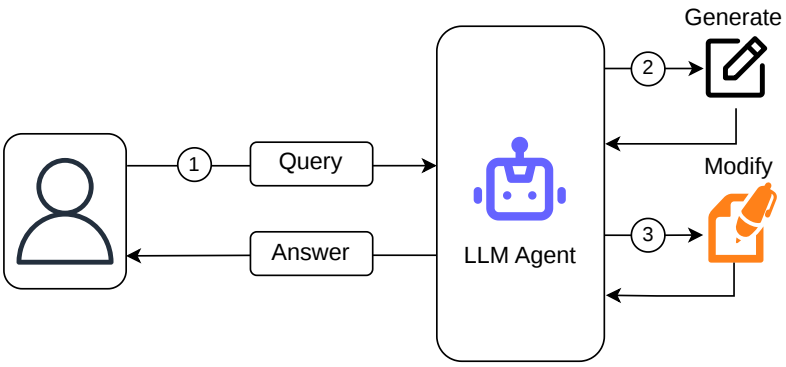

Figure 3.8: Template-Based YAML Generation

The previous approach, while does an effort to introduce more abstract, and structured generation, still allows free-form modification of the generated template in the second stage, which can introduce the possibility of deviating from the correct schema. To address this, the second variant, shown in Figure 3.8, modifies the generated configuration by invoking a parameter-modification function, shown in Figure 3.9, that allows targeted updates to specific fields within the YAML. The function enforces a static input format, where the model must provide:

- Explicit paths to the parameters to be updated within the YAML structure
- The corresponding new values to be assigned

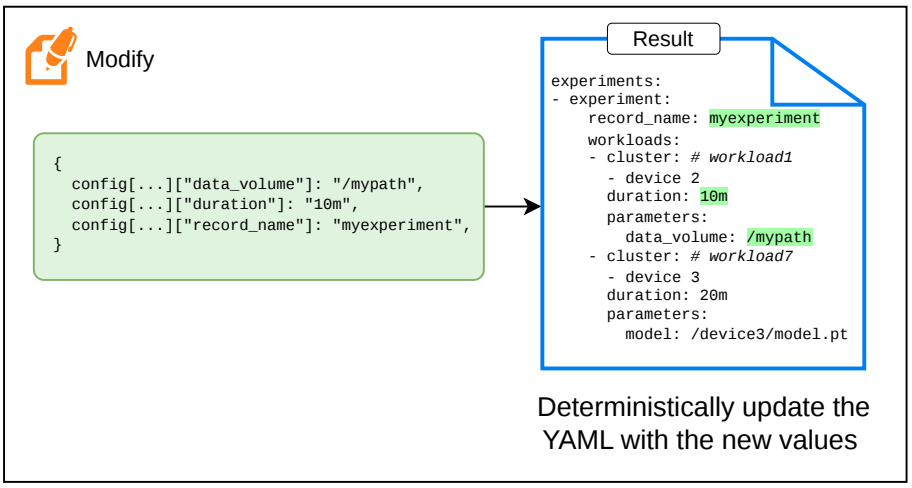

Figure 3.9: Template-Based YAML Modification Function

These paths reference nested fields within the configuration (e.g., workload parameters, experiment duration, or record identifiers). The function parses the previously generated YAML, applies the requested updates through a deterministic patching mechanism, and returns the updated configuration. To maintain schema integrity, only paths that already exist within the YAML are modified, and updated values are restricted to scalar types (int, char, string, bool) — composite types such as dicts, lists, or tuples are not permitted, as they can break the structure of the YAML by adding non-existent parameters or replacing and removing valid ones. If any modification fails — for example due to an invalid path or incompatible value — the function returns a detailed error message, prompting the model to retry. Here the purpose of the *current YAML configuration* variable is even more pronounced, as reliable transmission of the intermediate YAML configuration between these stages must be guaranteed — and storing it in the session memory ensures exactly that. Relying on the LLM to pass the YAML directly between functions would fundamentally undermine the purpose of the abstraction that we introduce, reintroducing the risk of free-form, unvalidated generation that the approach was specifically designed to avoid.

3.2.1.4 Template-based YAML generation with constrained decoding

The template-based YAML generation approach, while effective in abstracting a substantial portion of YAML generation, still appears to have two principal limitations that motivate a further refinement of the design. The first is a compliance problem. Even with a defined input format, which is a lot simpler and smaller than in free-form generation,

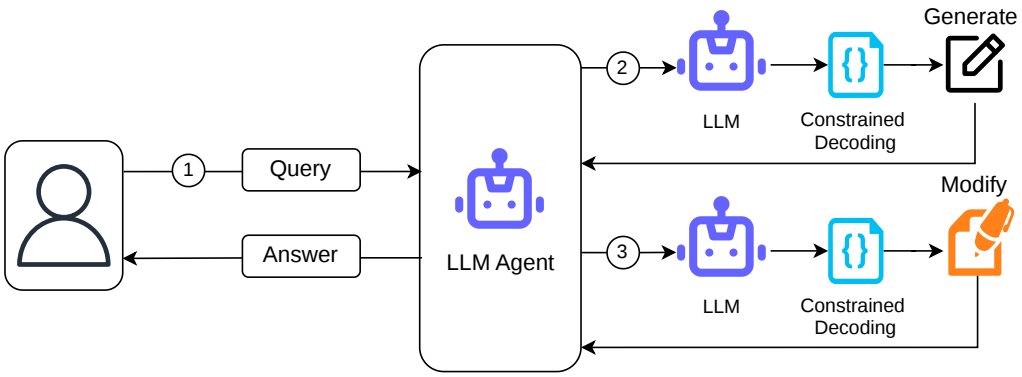

Figure 3.10: Template-Based Schema-Constrained YAML Generation

the agent is not guaranteed to adhere to it. When the model fails to produce a conforming input, the function rejects it and the model must re-invoke the tool, resulting in wasted computation, inflated token counts, and increased response latency — the latter being a particularly critical concern in interactive agentic systems where responsiveness directly affects usability. The second limitation is a broader problem of cognitive overload. Large language models, and in particular smaller and less capable ones, are prone to confusion when required to simultaneously satisfy multiple competing instructions. The smolagent framework [49] itself imposes a substantial system prompt that instructs the model on how to produce the correct format for tool calling, structure its responses, handle errors, and adhere to a range of framework-specific conventions. Beyond the system prompt, each individual tool carries its own detailed description, specifying input schemas, output formats, usage constraints, and expected behavior — all of which the model must internalize and comply with throughout the generation process. This accumulation of contextual information causes the effective token count to grow substantially before the model even gets to know its current task, making inference not only more expensive and computationally wasteful, but fundamentally more complex. The model is effectively asked to reason about the task while simultaneously managing a dense web of formatting and compliance requirements, a combination that increases the likelihood of errors, omissions, and schema violations.

One principled solution to both of these problems is to isolate tasks that require strict adherence to structured output formats into dedicated, self-contained inference calls, insulated from the broader agentic context. YAML configuration generation is precisely such a task, as its execution depends only on the user's input query, previous conversation exchange

in some cases, and the predefined workload schemas, while it carries no dependency on the agent's internal state or ongoing reasoning. It is therefore a natural candidate for delegation to a secondary, focused inference call. This isolation offers a significant additional advantage - it enables the use of constrained decoding on this specific subtask. Although the smolagent framework necessarily permits free-form generation throughout the agent loop — both to capture observations and to support flexible planning across heterogeneous tasks — this permissiveness is precisely what allows the model to fail schema compliance and trigger redundant function invocations. By performing an intermediate inference call dedicated solely to YAML generation, we can leverage the structured output functionality exposed by the OpenAI API, which accepts a Pydantic schema as a required output format and enforces it through constrained decoding at generation time. This guarantees, by construction, that the model's response always conforms to the specified schema, substantially reducing the need for complex error-handling and rollback mechanisms, minimizing the number of steps required to complete the task, and decreasing the total number of tokens generated across the interaction.

The constrained decoding approach is further strengthened in the parameter modification stage through the introduction of dynamic schema construction. Rather than asking the model to produce parameter paths and their new values — an approach susceptible to inconsistency, since the validity of a path cannot be verified at generation time — we instead use the assembled template produced during the first stage to dynamically generate a Pydantic schema at runtime. This schema fixes the overall structure of the YAML and exposes only the leaf-level parameter values as modifiable fields, preventing the model from introducing structural changes or referencing paths that do not exist. In this way, the modification stage enforces an even stricter generative constraint than the previous approach, further narrowing the space of possible outputs to only those that are semantically and structurally valid with respect to the configuration being refined.

It is important to note that this architecture does not necessarily require provisioning an entirely separate model instance. Since the agent loop is sequential rather than concurrent, the primary LLM model that acts as the agent is idle during tool execution — the agent pauses, and the LLM is not engaged in any ongoing inference. This creates an opportunity to reuse the same underlying model instance for the focused, isolated YAML generation call, without any interference with or disruption to the outer agentic loop. The result is a clean architectural separation achieved at no additional infrastructure cost. The agent handles high-level task orchestration and tool coordination, while the inner inference call

operates under a minimal, task-specific prompt optimized exclusively for structured configuration generation.

This design does, however, introduce additional considerations around context management and history transfer. Since the isolated inference call operates outside the agent's standard execution context, the relevant conversation history and task description must be explicitly passed to the model at invocation time. This is necessary to ensure the model has sufficient context to interpret the user's intent correctly — particularly in cases where the user refers implicitly to earlier parts of the conversation, such as requesting a modification to a configuration generated in a previous turn. Properly transferring this context is therefore an essential implementation concern, and one that must be handled carefully to preserve the coherence of the generation process across multi-turn interactions.

3.2.2 Automated Workload Deployment

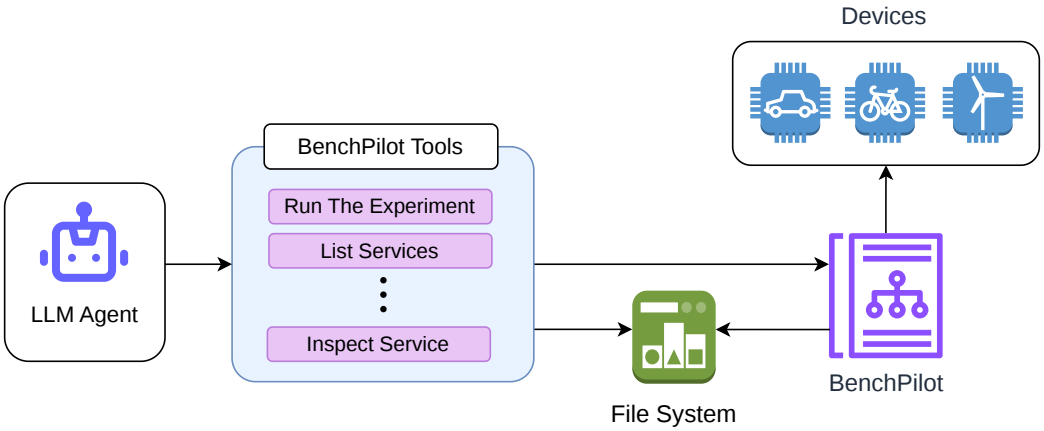

Figure 3.11: Interaction with BenchPilot through tools

Experiment deployment is another responsibility that can be delegated to BenchAgent. The user can make a natural language query to the BenchAgent to deploy an experiment, delegating the operational details to the agent. BenchAgent preserves full monitoring capability, keeping the user continuously informed of all actions taken on their behalf, as well as the real-time logs of running experiments, so the user remains fully aware of what is happening.

To deploy a generated YAML configuration, the agent is required to invoke a dedicated tool function, passing the filename of the target configuration as a parameter. This function

executes the necessary deployment commands natively supported by BenchPilot. Once the deployment command has been issued, a follow-up verification function is automatically triggered after a short delay of five seconds, allowing sufficient time for the system to initialize before the deployment status is assessed. This verification step confirms whether the experiment has been successfully launched or whether an error has been encountered at the deployment stage, which is them communicated to the user.

Throughout the execution of the experiment, the user is provided with real-time visibility into its progress through the system interface. The agent can invoke additional dedicated tools to continuously monitor the overall state of the experiment, for example tracking whether it remains active, has completed, or has transitioned into an error state. In the event that a failure is detected, the agent can inspect the associated error output, extracting and surfacing the relevant diagnostic information to the user.

3.2.3 Benchmarking Result Evaluation

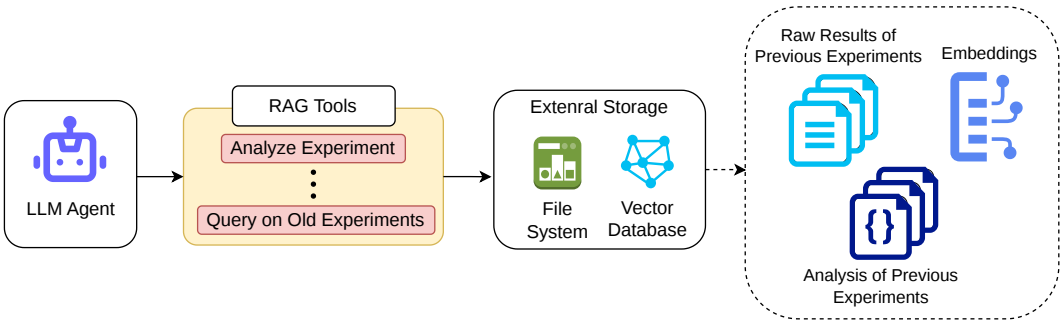

Figure 3.12: Evaluation of the benchmarking experiments through tools interacting with Vector Database and Storage System

Once the experiments have been completed, the user can request the agent to perform an analysis of the results. Such analysis, if performed manually, is very time consuming, while raw JSON outputs from the BenchPilot are hard to interpret. This is a multi-step process, and its design is informed directly by the underlying mechanics of BenchPilot itself. BenchPilot produces separate output files for each individual experiment run, even when multiple runs share the same configuration file. The results contained within these files can be substantial in size, and naively aggregating all experimental outputs into a single evaluation request would risk exceeding the context window of the underlying language model. Given that BenchPilot executes experiments sequentially rather than in parallel, each experiment can be evaluated independently, in the same order in which it

was executed.

For each experiment, a deterministic extraction process is applied to the BenchPilot-generated CSV files, retrieving relevant statistical measurements across all recorded metrics — including, among others, energy consumption, throughput, and resource utilization. Rather than exposing the raw numerical data directly to the language model, we compute a suite of descriptive statistics — mean, median, maximum, and minimum values — for each metric and each workload within the experiment, and present these in a structured, human-readable format. An example can be seen below:

```
REPORT
========================================
Total seconds: 550
Total workloads/metrics: 56

METRIC CATEGORY: NETWORK_SENT
----------------------------------------

Workload Worker: nc8_network_sent_bench-agent-jupyter
  Mean: 192.5700
  Median: 10.9999
  Std: 850.0714
  Min: 0.0000
  Max: 4561.3530
  First activity (sec): 0.0000
  Last activity (sec): 521.0000


Workload Worker: nc8_network_sent_benchpilot-agent-run
  Mean: 23088.5887
  Median: 31828.7900
  Std: 11328.8751
  Min: 0.0000
  Max: 31828.7900
  First activity (sec): 0.0000
  Last activity (sec): 521.0000

. . .
```

This preprocessing step is a deliberate design choice, since language models are known to exhibit weakness when confronted with large volumes of raw numerical data, often struggling to perform accurate arithmetic or draw reliable quantitative inferences from dense tabular content [78]. By distilling the measurements into a compact statistical summary before presenting them to the model, we substantially reduce this source of error and direct the model's reasoning capacity toward interpretation and synthesis rather than

computation. This structured statistical summary is complemented by the raw JSON files also produced by BenchPilot, which contain the log and error messages specific to each workload run. Together, these two sources of information provide the model with a comprehensive and balanced view of each experiment's execution and outcome.

Drawing on this combined context, the agent produces an evaluation report for each experiment . To ensure that these reports remain accessible and retrievable beyond the scope of the current conversation, each generated report is partitioned into fixed-length document chunks of one thousand characters, which are stored in a vector database. This enables retrieval of past evaluation results in future sessions, allowing the agent to draw conclusions on previous experiments.

Once evaluation reports have been produced and stored, the user can, at any point during the session, request the agent to retrieve and surface results from past experiments that are relevant to the current context, enabling a form of cross-referential analysis that would be laborious and error-prone if performed manually. This retrieval can be guided by a range of factors depending on the user's investigative focus: the user may wish to compare results across experiments that share the same physical devices, isolate runs that exercised the same workloads under different configurations, or identify prior experiments that produced similar outcomes or exhibited analogous failure patterns.

3.2.3.1 Knowledge Base

To store and manage the accumulated experimental results and agent-generated evaluation reports, we adopt a vector database as our primary knowledge persistence mechanism. This makes it a natural fit for a system in which the user may wish to retrieve past experiments based on loosely specified semantic cues — such as a particular performance characteristic, a class of error, or a general description of experimental conditions — rather than precise identifiers. In practice, we use **ChromaDB** as our vector database, storing both the raw BenchPilot-generated output, the structured evaluation reports produced by the agent, and their embeddings. To encode our data into the embeddings we use the **all-MiniLM-L6-v2** model [79, 80], a lightweight but highly capable sentence embedding model derived from knowledge distillation of larger transformer architectures. While compact in size, it produces 384-dimensional embeddings and offers a balance between representational quality and inference efficiency — an important consideration in a system where embedding operations are performed repeatedly across a growing corpus of experimental documents. To ensure that retrieval operates at an appropriate level of gran-

ularity and that individual chunks remain within the context capacity of the embedding model, documents are split into chunks of 1000 characters with an overlap of 200 characters between consecutive chunks. The overlap preserves local coherence across chunk boundaries, preventing the truncation of semantically connected content and ensuring that no meaningful context is lost at the point where one chunk ends and the next begins. Each chunk is stored alongside its embeddings and a set of metadata fields that record supplementary structural information — including the chunk index and the filename of the source document — enabling the system to maintain traceability across the fragmented representation of each report and to reconstruct the provenance of any retrieved passage.

3.2.3.2 Similarity Metrics

Retrieval from the vector database is performed using cosine similarity as the primary distance metric over the embedding space. Cosine similarity measures how similar two vectors are in a high-dimensional space, making it well-suited for comparing text embeddings. A cosine similarity of one indicates perfect alignment, and therefore maximal semantic relatedness, while a value of zero indicates orthogonality and semantic independence.

3.2.3.3 Retrieval Logic

To retrieve relevant documents for a given query, such as "*Are there any previous experiments on orin which had CPU utililazion on 99%?*", the system first embeds the query using the same **all-MiniLM-L6-v2** model [80] used during document ingestion, ensuring that query and document representations are directly comparable. A maximum of ten candidate documents are retrieved from the database based on their cosine similarity to the query embedding. However, returning a fixed number of results regardless of their actual relevance to the query is a known limitation of top-k retrieval strategies. In cases where the query is highly specific or the knowledge base is sparse, the tenth-ranked document may be semantically distant from the query and of little practical value. To address this, we introduce a dynamic relevance thresholding mechanism that filters the retrieved candidates based on the distributional characteristics of their similarity scores, rather than applying a globally fixed cutoff.

Concretely, given the set of cosine distances $d_1, d_2, \ldots, d_k$ returned for a query, the threshold τ is computed as:

$\tau = d_{\text{min}} + \beta \cdot (d_{\text{max}} - d_{\text{min}})$

where d_{min} and d_{max} denote the minimum and maximum distances in the retrieved set, and $\beta \in [0, 1]$ is a spread parameter that controls the selectivity of the filter. Only documents whose distance satisfies $d_i \leq \tau$ are retained and passed to the model as context, meaning when the retrieved documents are tightly clustered and uniformly relevant, the threshold is permissive and most candidates are retained; but when the scores are widely spread and drop off sharply, the threshold is correspondingly restrictive, pruning the less relevant tail of the result set. Setting $\beta = 0.5$ provides a balanced default that retains documents within the closer half of the similarity range, though this parameter can be adjusted.

This retrieval architecture supports the broader Retrieval-Augmented Generation (RAG) paradigm that underpins the system's analytical capabilities. By dynamically integrating retrieved experimental context into the model's generation process, RAG allows the agent to produce evaluations and comparisons that are grounded in the accumulated history of prior experiments, rather than relying solely on information present in the current conversation. This is the cornerstone for our task of cross-experiment result analysis, where the model must reason over results that were produced in earlier sessions and would otherwise be entirely inaccessible within the constraints of a finite context window.

4 Evaluation

4.1 Research Questions

The design and evaluation of BenchAgent are guided by a set of focused research questions that address both the methodological challenges of LLM-driven configuration generation and the broader system-level concerns of agentic deployment in a real-world benchmarking context.

RQ1 – How reliably can agentic LLM systems generate YAML configuration files for edge benchmarking, and how does reliability vary across generation strategies and model scales?

This question motivates the comparative evaluation of free-form, function-based, and constrained decoding approaches to YAML generation, and examines how the choice of strategy interacts with the instruction-following capability of the underlying model.

RQ2 – To what extent does increasing abstraction from free-form generation improve accuracy, token efficiency, and reduce the frequency of tool re-invocations caused by validation failures?

This question focuses specifically on the practical impact of replacing unconstrained generation with a structured, template-mediated approaches, and quantifies the cost reduction and reliability improvement associated with this architectural choice.

This question addresses the technical integration of constrained decoding within the smolagents framework, and examines the conditions under which this approach is both feasible and beneficial relative to softer enforcement mechanisms.

RQ3 – What are the practical trade-offs between deploying large and smaller, locally-deployed open-weight models in an agentic benchmarking context, and under what conditions can smaller models be made to perform reliably?

This question examines the cost-reliability frontier across the models evaluated in this work.

4.2 Evaluation Methodology

Evaluation of agentic systems is challenging, since there is no single standardized benchmark for evaluation, especially such system designs that are curated for specific services and built on top of existing infrastructure. Therefore we design our own datasets for evaluation of our pipeline, to analyze quantitatively the performance.

4.3 Testbed

4.3.1 Hardware

Our experimental infrastructure consists of two complementary hardware environments, each suited to different deployment scenarios. The first is a large local server equipped with 1 GPU with 15 GB VRAM and 95 GB of system RAM, which serves as the primary deployment target for lightweight, edge-compatible models. The second is a local High-Performance Computing (HPC) cluster equipped with 4 NVIDIA Tesla V100 GPUs, which provides the computational capacity required for hosting larger models that would otherwise exceed the memory constraints of the edge server. This dual-environment setup allows us to evaluate models across a spectrum of hardware capabilities, reflecting realistic deployment conditions in the edge continuum.

4.3.2 Models

We evaluate a diverse set of language models acting as agents within our pipeline, selected to cover a broad range of sizes, architectures, and deployment paradigms. The evaluation spans large and smaller locally-deployed open-weight models, enabling us to assess performance trade-offs across capability, resource efficiency, and deployment feasibility. Models are sourced from different providers — Ollama [81], and Unsloth [82] from HuggingFace [83] — depending on the target hardware environment and inference framework used. Model selection was guided by several considerations. We sought diversity in scale and architecture, including both dense transformer models and mixture-of-experts (MoE) architectures. The chosen models can be operated fully on-premise and are thus more suitable for edge or privacy-sensitive deployments, and thus quantization was a key consideration.

4.3.2.1 Model Serving and Inference

To support local LLM deployment across different hardware environments, we employ two complementary inference frameworks.

Ollama [81] provides a unified environment for running large language models locally, with built-in support for model management, versioning, and execution through a standardized OpenAI-compatible API. In our setup, Ollama is deployed on the large server, where it serves as the primary backend for executing LLM queries. The context window was fixed to 32k tokens for all models, corresponding to the smallest maximum context size among the evaluated models. This ensures a fair and controlled comparison, while also avoiding the significant inference-time and memory overhead introduced by larger context windows on the weaker hardware.

For resource-constrained environments, we additionally employ llama.cpp [31], a high-performance inference framework implemented in C/C++ that enables efficient model execution through techniques such as quantization and memory-efficient data structures, without requiring specialized accelerators. In our experiments, llama.cpp is deployed on the HPC server as a flexible alternative backend.

By leveraging both frameworks within a unified OpenAI-compatible interface, we ensure consistency across experimental conditions while benefiting from each framework's complementary strengths in usability, flexibility, and performance.

4.3.2.2 Qwen3 A32B

Qwen3 A32B is a 32-billion-parameter mixture-of-experts model in which approximately 3.3 billion parameters are active per inference step, making it significantly more efficient than equivalently-sized dense models [84]. It is designed for instruction-following tasks and is quantized to Q8_0 precision, which retains high output fidelity while reducing memory requirements sufficiently for deployment on the HPC server. The model is obtained from HuggingFace via the Unsloth provider, which offers optimized quantized checkpoints compatible with llama.cpp, the inference backend used on the HPC environment. Its MoE architecture and relatively high active parameter count make it a strong candidate for structured generation tasks, and we treat it as the primary representative of capable, locally-deployable models in our evaluation.

4.3.2.3 Granite 4 3B

Granite 4 3B is a compact, 3-billion-parameter model developed by IBM, fine-tuned from its base using a combination of open-source instruction datasets with permissive licenses and internally collected synthetic data [85]. A distinguishing feature of the Granite 4 series is its emphasis on instruction following and tool-calling capabilities, which are central requirements for agentic applications. These properties make it particularly relevant to our setting, where the model must reliably invoke tools and produce structured outputs within a constrained workflow. Given its small footprint, the model is hosted on the large server and served via Ollama, making it representative of the class of efficient, edge-deployable models. Its inclusion allows us to assess the lower end of the capability-efficiency spectrum and evaluate how well a highly constrained model can perform within an agentic pipeline when supported by structured decoding.

4.3.2.4 Ministral 3 8B

Ministral 3 8B is an 8-billion-parameter model developed by Mistral AI, designed specifically for edge deployment and capable of running on a wide range of hardware without requiring large-scale infrastructure [86]. A defining characteristic of the Ministral 3 series is its emphasis on agentic capabilities, offering native function calling and structured JSON output, which are central requirements for tool-driven workflows. The model further supports a 256k token context window, enabling it to handle long-horizon tasks and extended conversation histories that are common in complex agentic pipelines. Beyond its agentic strengths, the model supports vision inputs and multilingual inference across dozens of languages, broadening its applicability across diverse settings. In our deployment, the model is hosted on the large server and served via Ollama, and its permissive Apache 2.0 license ensures it can be freely used and modified in both research and commercial contexts. Its inclusion in our evaluation allows us to assess how a mid-scale, edge-optimised model with strong native tool-use capabilities performs within a structured agentic pipeline, positioning it as a representative of the efficiency-capability trade-off between the most constrained and most powerful models in our benchmark.

4.4 YAML Generation Accuracy

4.4.1 Dataset

We construct a dataset for evaluating the YAML generation capabilities of the agent. The dataset consists of 20 queries, formulated to be unambiguous, ensuring that the evaluation focuses on the agent's ability to satisfy the expressed requirements rather than its ability to interpret an unclear request. The queries vary in difficulty, where difficulty is defined by the number of explicit constraints or requirements the user expresses within the query. Simpler queries may contain only one or two constraints, while more complex ones require the agent to simultaneously satisfy a larger number of conditions, such as more workloads, and more parameters to change.

An example of what a query can be:

```
Create me a streaming mlperf workload using TensorFlow on rasp pi.
Change the model\_volume\_path to '/myfolder/model'.
```

The full dataset with the queries can be seen in Figure 4.1.

For each query, a corresponding ground-truth YAML file is hand-authored to represent the correct and complete satisfaction of all expressed constraints. The accuracy of the agent's generated YAML is then measured by comparing to the ground-truth configurations, with the scoring system explained below.

4.4.2 Accuracy Evaluation Pipeline

During the generation step, the agent-produced responses first pass through an initial validation filter. This stage performs structural validation of the generated YAML, verifies the presence of required top-level fields, such as the list of experiments and the inter-experiment idle time, and checks that all fields included in the configuration are valid and supported by at least one workload definition. The YAML configurations that survived this are stored in the session memory as the "current YAML configuration" variable, and enter this evaluation stage. Configurations that failed generation-time validation are assigned a score of zero without further analysis.

The evaluation of a generated YAML is performed through an hierarchical matching and scoring procedure operating at three levels: experiment alignment, workload alignment, and field-level comparison. In the sections below the scoring mechanism is described for each level.

4.4.2.1 Experiment Alignment

When both documents contain more than one experiment, the evaluator must determine which generated experiment corresponds to which ground-truth experiment before comparing their contents. This cannot be assumed to follow declaration order, as an LLM may produce experiments in a different sequence than the ground truth without this being semantically wrong.

The alignment is solved as an assignment problem over all permutations of generated experiments against ground-truth experiments. For each candidate pairing of a ground-truth experiment with a generated experiment, a preliminary similarity score is computed by running the workload matching procedure described in the following section and averaging the resulting field scores. The permutation that maximizes the total similarity across all experiment pairs is selected as the alignment. For a manageable number of experiments ($n \leq 6$), the search is exhaustive over all permutations; for larger counts, a greedy fallback is applied.

Critically, this alignment step operates only at the experiment level — it determines which generated experiment maps to which ground-truth experiment, but it does not move workloads between experiments. The constraint that each workload belongs to a specific experiment is preserved throughout the evaluation.

4.4.2.2 Workload Matching

Within each aligned experiment pair, the evaluator must match individual workloads from the generated document to those in the ground truth. A single experiment may contain multiple workloads of the same type — for example, two mlperf workloads that differ only in their execution mode — so matching cannot rely on workload names alone.

Matching is performed greedily. For each ground-truth workload, all unmatched generated workloads are scored as candidate pairs and the one with the highest field-level score is selected as the match. This process repeats until all ground-truth workloads have been matched or no generated workloads remain. If the generated document contains fewer workloads than the ground truth, unmatched ground-truth workloads are compared against an empty dictionary, meaning all of their fields are recorded as misses.

4.4.2.3 Field-Level Comparison

Each matched workload pair is scored by flattening both workloads into sets of dot-separated key–value paths and comparing them field by field. More specifically, nested dictionary structures, are expanded into dot-separated key paths. This means that "parameters.model_file" is treated as a distinct comparable unit rather than subsumed into a single comparison of the entire "parameters" block. The consequence is that a workload which gets nine out of ten parameters correct receives a substantially higher score than one that gets only one correct, reflecting the genuine partial quality of the output.

For each aligned workload pair, a field-level comparison is performed over all ground-truth-defined keys. Let TP denote the number of correctly matched fields, FP the number of extra fields present in the generated workload but not in the ground truth, and FN the number of missing ground-truth fields or wrong-value fields. We compute precision and recall over flattened configuration fields and define the workload-level score using the F1 measure:

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$\text{WorkloadScore} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

This F1 formulation penalises both omitted ground-truth fields (low recall) and any generated field that does not exactly match the ground truth — whether hallucinated or incorrectly valued (low precision).

The score of an experiment is the arithmetic mean of the workload scores of all its matched workloads. The overall score for a complete YAML is the arithmetic mean of all workload scores across all experiments, assigning equal weight to each workload irrespective of which experiment it belongs to. All scores lie in [0,1], where 1 denotes a perfect match in both structure and field values.

4.4.3 Task Completion Rate

Task completion rate measures the proportion of queries for which the agent reaches a successful conclusion without being interrupted by the maximum retry limit, which was set to 10 across all the experiments. In the SmolAgents framework, a task is considered complete when the agent explicitly invokes the built-in *final_answer* tool, which signals

that it has finished reasoning and is returning its output to the user. A task is considered incomplete if the agent exhausts its allowed retry budget before invoking the final_answer, regardless of any partial progress made during the intermediate steps. This metric therefore captures the agent's basic reliability and its ability to converge on a solution within the constraints of the agentic loop, independently of the quality of the output it produces. Results for the task completion rates can be found in Tables 5.2, 5.3, 5.1.

4.4.4 Number of Tool Invocations per Task

As the agent generates an answer for a single query, it will invoke a number of tools — such as database retrieval, or BenchPilot execution tools. For each query, we record the total number of tool calls made by the agent and report the mean across the dataset. More specifically, we report the mean number of invocations for each tool category:

- **RAG Tool Invocations**, which include any interaction with the vector database.
- **BenchPilot Tools**, corresponding to the tools that call BenchPilot commands.
- **Final Answer**, which corresponds to the built-in function invoked by the agent once the task is completed and the final result is returned to the user.
- **Tool Call Failure**, which captures the number of failed tool invocations, either due to incorrect parameter formatting by the agent or errors raised during tool execution.
- **YAML Creation Tools**, which correspond to the set of tools described earlier that are responsible for constructing the YAML configurations.

This metric reflects both the efficiency of the agent and its reliance on external context and tooling to complete the task. Queries requiring more complex configurations are expected to result in a higher number of tool calls, while high variance across queries of similar difficulty may indicate inconsistent planning behaviour. As our dataset focuses only on the task of YAML creation, the expected tool groups to be called are only the "YAML creation Tools", and "Final Answer". The Figures 5.2, 5.1, 5.3 illustrate the results.

4.4.5 Computational Cost Analysis

We record the total number of tokens produced by the agent per query, including both reasoning tokens and the final YAML output. This metric serves as a proxy for computational cost and verbosity. An agent that generates substantially more tokens than necessary to produce a correct YAML may be engaging in redundant reasoning or producing verbose

intermediate outputs. Total token count is reported as a mean across the dataset. Tables 5.2, 5.3, 5.1 illustrate the results.

4.4.6 Latency per Task

Latency is defined as the wall-clock time from when a query is submitted to the agent to when a complete response is returned. This measure includes all time spent on internal reasoning, tool usage, and response generation. Reported latency is summarized as the mean and standard deviation across all queries in the dataset. Results are also found in the Tables 5.2, 5.3, 5.1.

Table 4.1: Dataset Prompts

ID	Prompt
1	Create me 2 mlperf ONNX workloads on xavier in one experiment, one batch mode and the other streaming mode. The streaming one should have the model file cifar10.onnx.
2	Colocate two mlperf streaming workloads on xavier, one using ncnn and one using onnxruntime.
3	Create one batch mlperf workload using TensorFlow and one simple CPU stress workload. Put them in the same experiment on raspberry pi.
4	Create me one mlperf ONNX streaming workload and one database workload. Put them in separate experiments on orin.
5	Create two mlperf workloads where both use NCNN models - first batch, second streaming. Make the accuracy enabled.
6	Create me a marketing-campaign workload using Flink and an mlperf TensorFlow batch workload. Put them in one experiment on xavier.
7	Create one streaming NCNN mlperf workload on orin with 4 worker threads.
8	Create me two simple workloads: one IO stress and one VM stress. Put them in one experiment on jetson.
9	Create a streaming mlperf workload using NCNN on nc8. Use dataset 'ncnn_dataset'. For output volume use the path '/output_volume_test'.
10	Create me one batch tensorflow mlperf workload and one marketing-campaign workload using Spark, all in one experiment on xavier.
11	Create two mlperf workloads, one ONNX and the other TensorFlow, both batch. Put each in separate experiments on orin.
12	Create one streaming TensorFlow mlperf workload on xavier and one simple IO stress workload on orin. Put both workloads on the same experiment.
13	Create a config with one experiment on xavier with one database workload with duration of 10 minutes, and one batch mlperf ONNX workload with duration of 5 minutes.
14	Create me a streaming mlperf workload using TensorFlow on rasp pi. Change the model_volume_path to '/myfolder/model'.
15	Create two experiments on jetson, each with one mlperf ONNX workload. Set the accuracy as true.
16	Create me one simple iperf3 server workload on orin. Its ip should be 10.20.20.20.
17	Create an iperf3 client workload targeting server ip 10.0.0.1 on xavier.
18	Create me a streaming mlperf workload with model file mobilenet.onnx on xavier.
19	Create a marketing-campaign workload using Storm with 8 ackers, on xavier.
20	Create me three workloads in one experiment: one batch ONNX mlperf, one streaming NCNN mlperf, and one CPU stress, all deployed on rasp pi.

5 Results / Findings

5.1 Overall Accuracy

Across all three models, the constrained decoding approach (*template-constr-dec*) consistently achieves the highest field-level accuracy, attaining scores of 0.970, 0.846, and 0.951 for Qwen, Granite, and Ministral respectively. Free-form generation (*free-form*) is the weakest approach in all cases, with scores of 0.767, 0.101, and 0.336. The two intermediate approaches — *template-free-form* and *template* — occupy the space between these extremes, though their relative ordering is not consistent across models, which suggests that the benefit of each design choice is not uniform and interacts significantly with the underlying model capability.

The most striking observation is the magnitude of the spread between best and worst approach as a function of model strength. For Qwen, the difference between the highest and lowest score is approximately 0.2. For Granite and Ministral, this gap widens to 0.74 and 0.62 respectively. This indicates that while a capable model such as Qwen can partially compensate for a weaker generation strategy, less capable models are far less accurate in generation without any structural constraints. In other words, the design of the generation strategy matters most precisely in the cases where it is needed most — when the model's inherent capacity for schema compliance is limited.

5.2 The Effect of Constrained Decoding

The *template-constr-dec* approach outperforms *template* — its closest structural counterpart — by a consistent margin across all models: +0.15 for Qwen, +0.46 for Granite, and +0.13 for Ministral. This improvement is attributable directly to the enforcement of schema compliance at decoding time. In the *template* approach, the model must navi-

Table 5.1: Average metrics per run - Qwen

	Steps	Latency (s)	Tokens	Score	Completion Rate (%)
template-constr-dec	4.5	31.4	22017.0	0.970	100.0
template	4.8	13.4	22227.6	0.817	100.0
template-free-form	3.9	15.1	17875.7	0.884	100.0
free-form	4.4	20.1	28607.5	0.767	100.0

Table 5.2: Average metrics per run - Granite

	Steps	Latency (s)	Tokens	Score	Completion Rate (%)
template-constr-dec	8.8	37.8	45192.8	0.846	30.0
template	9.1	24.9	46509.6	0.383	30.0
template-free-form	9.0	19.2	39394.8	0.200	35.0
free-form	11.0	119.9	113531.4	0.101	0.0

Table 5.3: Average metrics per run - Ministral

	Steps	Latency (s)	Tokens	Score	Completion Rate (%)
template-constr-dec	9.2	190.4	63038.2	0.951	40.0
template	10.4	183.9	70165.6	0.818	20.0
template-free-form	7.9	140.1	49776.2	0.896	65.0
free-form	9.7	245.1	90131.6	0.336	30.0

gate explicit path references into the YAML structure, a task that requires it to internalize the configuration layout and produce syntactically valid update specifications — a source of error even for capable models. In *template-constr-dec*, this burden is lifted: the dynamically constructed Pydantic schema exposes only the leaf-level fields as modifiable targets, making schema-invalid outputs structurally impossible. The result is not merely a marginal improvement in compliance, but a qualitative shift in reliability, particularly for Granite, where the gap between the two approaches is nearly half a point.

Notably, this gain in accuracy comes at a moderate token cost. Qwen consumes slightly fewer tokens under *template-constr-dec* than under *template* (22,017 vs. 22,227), while Granite and Ministral show a similar modest reduction (45,192 vs. 46,509, and 63,038 vs. 70,165 respectively). This suggests that the isolated inference call introduced by constrained decoding does not introduce meaningful token overhead, and in some cases may reduce it by preventing repeated failed modification attempts that consume tokens without advancing the task.

5.3 Free-form Generation as a Baseline

Free-form generation performs worst in terms of both accuracy and token consumption across all models. The token counts are substantially higher than all template-based variants: Qwen generates 28,607 tokens on average, Granite 113,531, and Ministral 90,131 — considerably more than any structured alternative. This is consistent with the hypothesis

that without structural scaffolding, the model must regenerate the full YAML configuration iteratively in response to validator feedback, a process that is both expensive and unreliable. The validator provides error messages that guide refinement, but for weaker models this feedback loop appears to be largely ineffective: Granite achieves only a score of 0.101, suggesting it is frequently unable to translate error messages into correct structural repairs, resulting in token-intensive cycles of invalid generation. This issue is particularly critical in edge deployment scenarios, where such smaller models are typically intended to operate under strict latency, memory, and compute constraints, making inefficient iterative generation both impractical and misaligned with their intended use cases.

Granite's free-form run is also the only configuration where the completion rate drops to 0%, meaning the model exhausted its step budget on every single task without successfully invoking the *final_answer* tool. This is compounded by the exceptionally high latency of 119.9 seconds per task, by far the highest across all model-approach combinations for Granite. Together, these figures paint a picture of a model trapped in a validation loop it cannot escape: it generates, fails validation, regenerates, fails again, and eventually hits the step limit without converging on a valid configuration. This failure mode is precisely the one that template-based approaches were designed to prevent, and the results validate that motivation.

5.4 Template-Free-Form Token Efficiency

The *template-free-form* approach exhibits an interesting duality across models. For Qwen, it achieves the lowest token count of any approach (17,875) while delivering a competitive score of 0.884 — second only to constrained decoding. For Ministral, it similarly achieves the lowest token count (49,776) with a respectable score of 0.896, and also achieves the highest completion rate of any Ministral configuration at 65%, indicating that the model frequently completes the task within budget. However, for Granite, the same approach collapses to a score of 0.200, the second-worst result overall.

This divergence suggests that *template-free-form* is an efficient and effective strategy when the model is sufficiently capable of performing free-form modification over a pre-structured template, but degrades sharply when model capability falls below a certain threshold. The template provides structural scaffolding for the initial configuration, but the subsequent free-form modification stage reintroduces the same risks as unconstrained generation — the model must produce a valid YAML in its entirety with modifications

applied, which requires maintaining structural coherence across the full document. For Granite, this step appears to be where the reliability breaks down.

5.5 Computational Cost and Latency

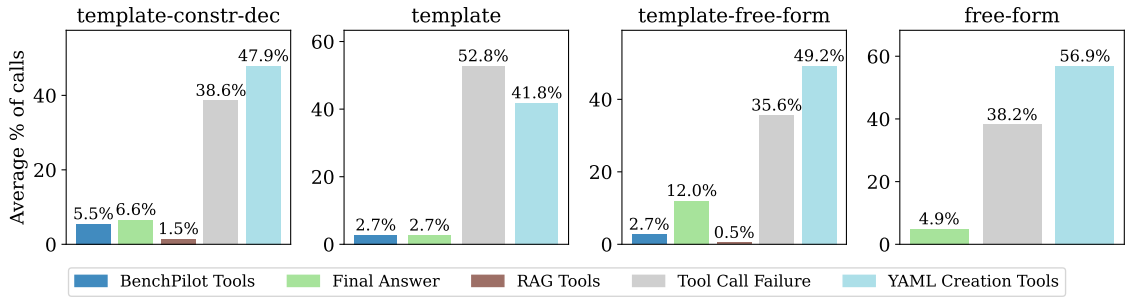

Figure 5.1: Tool Invocation Breakdown in Ministral

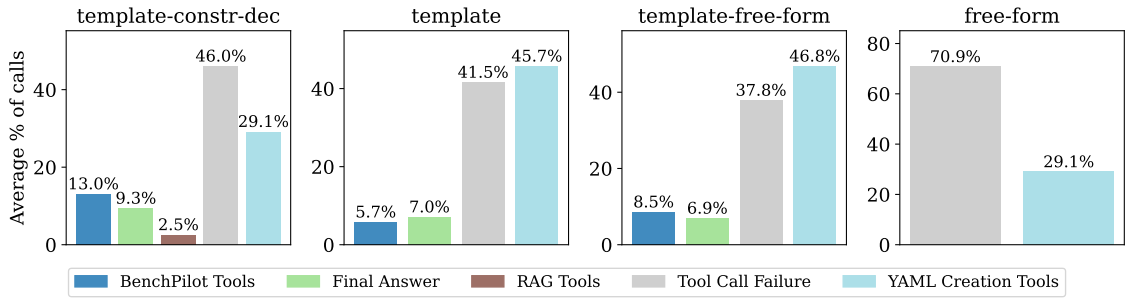

Figure 5.2: Tool Invocation Breakdown in Granite

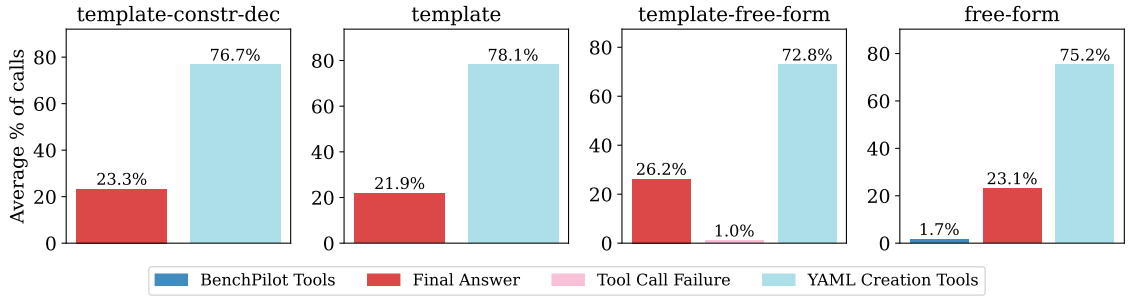

Figure 5.3: Tool Invocation Breakdown in Qwen

Latency results reveal an important practical consideration regarding constrained decoding. For Qwen, the *template-constr-dec* approach incurs the highest latency of all approaches at 31.4 seconds, compared to 13.4 and 15.1 seconds for *template* and *template-free-form* respectively. This overhead is attributable to the isolated inference call that

performs YAML generation under constrained decoding, which introduces an additional round-trip that is not present in other approaches.

For Granite, *free-form* is by a substantial margin the most expensive approach at 119.9 seconds, nearly five times the latency of *template-free-form* at 19.2 seconds. This further underscores the cost of the unbounded validation-retry loop that free-form generation enables for weaker models. The constrained decoding approach imposes a slightly higher latency through the isolated inference call, but substantially improves the performance.

5.6 Task Completion and Step Budget

The completion rate — the proportion of tasks in which the agent correctly invoked *final_answer* before exhausting its step budget — reveals significant differences in task completion reliability across models and approaches. Qwen completes its responses on 100% of tasks across all approaches, indicating that it reliably converges on a solution regardless of the generation strategy employed. This is consistent with its uniformly strong scores and relatively stable step counts (3.9–4.8 steps), suggesting that Qwen's instruction-following and schema compliance capacity is robust enough to succeed under any of the evaluated conditions.

Granite presents the starkest picture. It completes the responses on 0% of tasks under free-form generation, rising to only 30–35% under template-based approaches. Even with the best-performing strategy, the majority of Granite's tasks exhaust the step budget. The step counts corroborate this: Granite requires 8.8–11.0 steps depending on the approach, consistently approaching or reaching the maximum of 10 (with the 11th step reserved for the final answer). The *template-constr-dec* approach, despite achieving the highest score for Granite, still only completes 30% of tasks within budget — suggesting that while constrained decoding substantially improves the quality of outputs when the task does complete, it does not resolve the more fundamental issue of Granite's difficulty managing the agentic loop's coordination requirements. As a result, even when the task has effectively been completed and the tool has already returned the generated YAML configuration, the model fails to recognize that the objective has been achieved. Instead, it continues to re-invoke tools and repeatedly regenerate the YAML output.

Ministral shows the most variable completion rate behavior, ranging from 20% under *template* to 65% under *template-free-form*. However, the high termination rate under *template-free* form (65%) does not correspond to the highest score — this is achieved

instead by *template-constr-dec* (0.951). This apparent paradox may reflect that while Ministral converges faster under *template-free-form*, it does so at the cost of output quality, producing configurations that are structurally plausible but field-level incorrect. Under constrained decoding, the model requires more steps but produces more accurate outputs, suggesting that the additional steps correspond to meaningful refinement rather than unproductive looping.

5.7 Tool Invocations

The tool invocation breakdown reveals a stark contrast between Qwen and the two weaker models. Since the evaluated tasks required only YAML creation tools and a single final answer invocation to complete successfully, all other tool calls — BenchPilot tools, RAG tools, and tool call failures — represent erroneous or unproductive agent behavior. As shown in Figure 5.3, Qwen exhibits a highly focused invocation profile across all generation strategies, with YAML creation tools consistently accounting for 72–78% of all calls and erroneous invocations remaining negligible across all approaches. This confirms that Qwen operates efficiently and reliably regardless of the generation strategy, spending the vast majority of its step budget on productive work rather than error recovery or spurious tool calls.

For Granite and Ministral, the picture is considerably less favorable. Tool call failures constitute a substantial and often dominant share of total invocations, particularly under free-form generation, where they account for 56.9% and 70.9% of all calls respectively. This directly reflects the validation-retry loop described earlier: the model repeatedly produces invalid outputs, receives error feedback, and re-invokes the tool without successfully converging. Under template-based approaches the failure rate decreases, but remains high — for Granite, tool call failures still represent 37.8–46.0% of invocations even with structured generation strategies, indicating that schema compliance failures persist beyond the YAML generation stage and affect the broader tool-calling behavior of the agent.

An interesting pattern emerges when examining the final answer invocation rate across approaches. In both Ministral and Granite, `template-free-form` tends to produce a higher proportion of final answer calls than `template-constr-dec`, despite the latter achieving substantially higher accuracy scores. This indicates that under `template-free-form`, the models are terminating earlier and more frequently — consistent with the higher completion rates reported in the summary tables — but doing so with lower-quality outputs.

The models appear to converge on a conclusion faster, but at the cost of correctness. Constrained decoding, by contrast, directs more of the step budget toward productive YAML generation, takes longer to reach the final answer, but produces measurably more accurate configurations. This dissociation between termination rate and output quality highlights an important subtlety: early convergence is not necessarily a sign of successful task completion, and termination rate alone is an insufficient proxy for agent reliability.

5.8 Summary

Taken together, the results support several conclusions. First, constrained decoding is the most reliable generation strategy across all evaluated models and metrics, consistently delivering the highest accuracy with competitive or lower token consumption than unconstrained alternatives. Second, the performance gap between approaches is strongly mediated by model capability: weaker models benefit disproportionately from structural constraints, while more capable models are less sensitive to the choice of generation strategy. Third, free-form generation is both the most expensive and least reliable approach for weaker models, frequently resulting in token-intensive validation loops from which the model cannot recover within its step budget. Fourth, template-based generation without constrained decoding offers a meaningful intermediate improvement over free-form generation, but the residual freedom in the modification stage remains a source of error that constrained decoding eliminates. Finally, the latency overhead introduced by the isolated constrained-decoding inference call is modest and, in cases involving weaker models, is more than offset by the reduction in failed agent steps and their associated token and time costs.

6 Conclusion and Future Work

6.1 Conclusions

In this work we investigate whether an LLM-driven agentic system could meaningfully automate the experimental lifecycle of an edge benchmarking framework, reducing the manual overhead associated with configuration authoring, experiment deployment, and post-experiment analysis while remaining accessible through a natural language interface. To this end, we designed, implemented, and empirically evaluated BenchAgent, an agentic system built on top of the BenchPilot edge benchmarking framework, and conducted a systematic comparative study of the design choices that govern its core generation pipeline.

The empirical results support the central thesis that agentic LLM systems can automate structured configuration generation with high reliability, provided that appropriate constraints are placed on the generation process. Across all three evaluated models, the constrained decoding approach consistently achieved the highest field-level accuracy, with scores of 0.970, 0.951, and 0.846 for Qwen, Ministral, and Granite respectively, while remaining competitive in terms of token consumption and end-to-end latency. Free-form generation, by contrast, proved both the most expensive and least reliable strategy, particularly for weaker models, where it produced token-intensive validation loops from which the agent frequently could not recover within its allotted step budget.

A further finding of practical significance is that the performance gap between generation strategies is strongly mediated by the underlying model's instruction-following capacity. For a sufficiently capable model such as Qwen, all four strategies achieve acceptable results, with the worst-performing approach still reaching a score of 0.767. For weaker models such as Granite, the same spread widens to over 0.74, meaning that the choice of generation strategy is effectively the dominant factor in system performance. This has an important practical implication: in deployment settings where only smaller, locally hosted models are available — as is common in edge computing environments due to resource constraints — the design of the generation pipeline carries considerably more weight than in cloud-based settings where large frontier models are accessible.

The dynamic relevance thresholding mechanism introduced for RAG-based retrieval similarly demonstrated that adapting retrieval behavior to the distributional properties of individual queries, rather than applying a globally fixed selection policy, improves the pre-

cision of context injection without degrading recall on queries that admit multiple relevant documents. The layered memory architecture, separating high-level conversational context from low-level task execution history and integrating short-term in-context storage with long-term vector database-backed memory, provided a principled foundation for coherent multi-turn interaction within the constraints of finite context windows — a requirement that is particularly acute in agentic systems where the accumulation of tool call history and intermediate observations can rapidly exhaust available context capacity. Taken together, these contributions represent, to the best of our knowledge, the first integration of an agentic LLM system into an edge benchmarking framework that spans the complete experimental workflow from natural language query to post-experiment comparative analysis, and the first systematic empirical comparison of structured configuration generation strategies in this setting.

6.2 Limitations

Several limitations of the present study should be acknowledged, as they bear on the interpretation and generalizability of the results.

6.2.1 Scope of Evaluation Models.

The comparative evaluation of generation strategies was conducted across three models — Qwen, Granite, and Ministral — selected to represent a range of capability levels available for local deployment. While this selection provides meaningful coverage of the small-to-medium model segment, it does not include frontier-scale models such as GPT-4 or Claude, nor does it include models fine-tuned specifically for structured output generation or code synthesis. The degree to which the observed rankings between generation strategies hold for larger or domain-specialized models remains an open question.

6.2.2 Dataset Size and Diversity.

The evaluation dataset, while structured along multiple query scope axes, is necessarily limited in size and grounded in the specific workload types and configuration patterns supported by BenchPilot at the time of this study. The dataset does not cover all possible configuration combinations, and the results may not fully generalize to configurations of substantially greater complexity or to workload types that were not represented in the evaluation set. A larger, more diverse evaluation corpus would provide a more robust

empirical foundation for the conclusions drawn.

6.3 Future Work

The findings and limitations of this study suggest several directions for future research.

6.3.1 Fine-tuning for Structured Configuration Generation.

The results demonstrate that model capability is the primary moderating factor in generation strategy effectiveness. A natural follow-up is to investigate whether fine-tuning a smaller model on a curated dataset of BenchPilot configurations — or on structured YAML generation more broadly — can narrow the performance gap between smaller models and the constrained decoding approach, potentially yielding a more computationally efficient system that does not depend on API-level constrained decoding support. This would be particularly valuable for fully offline edge deployments where API-based inference backends are unavailable.

6.3.2 Generalization to Other Configuration Domains.

A direct extension of this work is to evaluate the generation strategies and architectural components developed here in other structured configuration domains — such as Kubernetes manifests, Ansible playbooks, or CI/CD pipeline definitions — where the same tension between generative flexibility and schema compliance arises. Such a study would establish the extent to which the conclusions drawn here are specific to the BenchPilot setting or reflect more general principles of LLM-driven structured generation.

6.3.3 Expanded Memory and Cross-Session Learning.

The layered memory architecture implemented in BenchAgent provides the structural foundation for cross-session knowledge accumulation, but the extent to which accumulated experimental history improves generation quality and retrieval precision over time was not the primary focus of this study. Future work should investigate the long-term dynamics of the memory system: how retrieval quality evolves as the experimental knowledge base grows, whether the dynamic thresholding mechanism remains well-calibrated as the vector store scales, and what strategies are most effective for managing knowledge base staleness as the BenchPilot framework evolves and older configurations become outdated.

BIBLIOGRAPHY

[1] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, p. 78–81, May 2016. [Online]. Available: https://doi.org/10.1109/MC.2016.145

[2] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020, pp. 42–54.

[3] M. Symeonides, D. Trihinas, G. Pallis, M. D. Dikaiakos, C. Psomas, and I. Krikidis, "5g-slicer: An emulator for mobile iot applications deployed over 5g network slices," in *2022 IEEE/ACM Seventh International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2022, pp. 115–127.

[4] B. Varghese, N. Wang, D. Bermbach, C.-H. Hong, E. de Lara, W. Shi, and C. Stewart, "A survey on edge performance benchmarking," 2020. [Online]. Available: https://arxiv.org/abs/2004.11725

[5] S. Ceesay, A. Barker, and B. Varghese, "Plug and play bench: Simplifying big data benchmarking using containers," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 2821–2828.

[6] F. Nikolaidis, A. Chazapis, M. Marazakis, and A. Bilas, "Frisbee: A suite for benchmarking systems recovery," in *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, 2021, pp. 18–24.

[7] J. Georgiou, M. Symeonides, M. Kasioulis, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Benchpilot: Repeatable reproducible benchmarking for edge micro-dcs," in *2022 IEEE Symposium on Computers and Communications (ISCC)*, 2022, pp. 1–6.

[8] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte,

I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[9] C. Shorten, C. Pierse, T. B. Smith, E. Cardenas, A. Sharma, J. Trengrove, and B. van Luijt, "Structuredrag: Json response formatting with large language models," 2024.

[Online]. Available: https://arxiv.org/abs/2408.11061

[10] Z. R. Tam, C.-K. Wu, Y.-L. Tsai, C.-Y. Lin, H. yi Lee, and Y.-N. Chen, "Let me speak freely? a study on the impact of format restrictions on performance of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2408.02442

[11] Y. Li, R. Ramprasad, and C. Zhang, "A simple but effective approach to improve structured language model output for information extraction," 2024. [Online]. Available: https://arxiv.org/abs/2402.13364

[12] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," 2024. [Online]. Available: https://arxiv.org/abs/2402.01680

[13] S. Chen, Y. Liu, W. Han, W. Zhang, and T. Liu, "A survey on llm-based multi-agent system: Recent advances and new frontiers in application," 2025. [Online]. Available: https://arxiv.org/abs/2412.17481

[14] A. Plaat, M. Van Duijn, N. Van Stein, M. Preuss, P. Van der Putten, and K. J. Batenburg, "Agentic large language models, a survey," *Journal of Artificial Intelligence Research*, vol. 84, Dec. 2025. [Online]. Available: http://dx.doi.org/10.1613/jair.1.18675

[15] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022. [Online]. Available: https://arxiv.org/abs/2205.01068

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[17] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[18] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[19] F. Wang, Z. Zhang, X. Zhang, Z. Wu, T. Mo, Q. Lu, W. Wang, R. Li, J. Xu, X. Tang, Q. He, Y. Ma, M. Huang, and S. Wang, "A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness," 2024. [Online]. Available: https://arxiv.org/abs/2411.03350

[20] C. Shi, H. Yang, D. Cai, Z. Zhang, Y. Wang, Y. Yang, and W. Lam, "A thorough examination of decoding methods in the era of llms," 2024. [Online]. Available: https://arxiv.org/abs/2402.06925

[21] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous, "Is temperature the creativity parameter of large language models?" 2024. [Online]. Available: https://arxiv.org/abs/2405.00492

[22] S. Geng, M. Josifoski, M. Peyrard, and R. West, "Grammar-constrained decoding for structured nlp tasks without finetuning," 2024. [Online]. Available: https://arxiv.org/abs/2305.13971

[23] B. T. Willard and R. Louf, "Efficient guided generation for large language models," 2023. [Online]. Available: https://arxiv.org/abs/2307.09702

[24] N. Jegham, M. Abdelatti, C. Y. Koh, L. Elmoubarki, and A. Hendawi, "How hungry is ai? benchmarking energy, water, and carbon footprint of llm inference," 2025. [Online]. Available: https://arxiv.org/abs/2505.09598

[25] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," 2024. [Online]. Available: https://arxiv.org/abs/2306.00978

[26] K. Egashira, M. Vero, R. Staab, J. He, and M. Vechev, "Exploiting llm quantization," 2024. [Online]. Available: https://arxiv.org/abs/2405.18137

[27] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm.int8(): 8-bit matrix multiplication for transformers at scale," 2022. [Online]. Available: https://arxiv.org/abs/2208.07339

[28] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020. [Online]. Available: https://arxiv.org/abs/2004.09602

[29] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2017. [Online]. Available: https://arxiv.org/abs/1712.05877

[30] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," 2023. [Online]. Available: https://arxiv.org/abs/2210.17323

[31] G. Gerganov and contributors, "llama.cpp: Llm inference in c/c++," https://github.com/ggml-org/llama.cpp, 2023, accessed: 2026-04-13.

[32] P. Liu, Z. Liu, Z.-F. Gao, D. Gao, W. X. Zhao, Y. Li, B. Ding, and J.-R. Wen, "Do emergent abilities exist in quantized large language models: An empirical study," 2023. [Online]. Available: https://arxiv.org/abs/2307.08072

[33] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," 2023. [Online]. Available: https://arxiv.org/abs/2307.03172

[34] F. Barbero, Álvaro Arroyo, X. Gu, C. Perivolaropoulos, M. Bronstein, P. Veličković, and R. Pascanu, "Why do llms attend to the first token?" 2025. [Online]. Available: https://arxiv.org/abs/2504.02732

[35] Y. Li, B. Dong, C. Lin, and F. Guerin, "Compressing context to enhance inference efficiency of large language models," 2023. [Online]. Available: https://arxiv.org/abs/2310.06201

[36] X. Wang, Z. Chen, Z. Xie, T. Xu, Y. He, and E. Chen, "In-context former: Lightning-fast compressing context for large language model," 2024. [Online]. Available: https://arxiv.org/abs/2406.13618

[37] Y. Wang, D. Krotov, Y. Hu, Y. Gao, W. Zhou, J. McAuley, D. Gutfreund, R. Feris, and Z. He, "M+: Extending memoryllm with scalable long-term memory," 2025. [Online]. Available: https://arxiv.org/abs/2502.00592

[38] N. Bui, S. Sharma, S. Lamba, S. Mishra, and R. Ying, "Cache what lasts: Token retention for memory-bounded kv cache in llms," 2026. [Online]. Available: https://arxiv.org/abs/2512.03324

[39] P. Chhikara, D. Khant, S. Aryan, T. Singh, and D. Yadav, "Mem0: Building production-ready ai agents with scalable long-term memory," 2025. [Online].

Available: https://arxiv.org/abs/2504.19413

[40] M. A. Haque, F. Rahman, K. D. Gupta, K. Shujaee, and R. George, “Tinyllm: Evaluation and optimization of small language models for agentic tasks on edge devices,” 2025. [Online]. Available: https://arxiv.org/abs/2511.22138

[41] P. Belcak, G. Heinrich, S. Diao, Y. Fu, X. Dong, S. Muralidharan, Y. C. Lin, and P. Molchanov, “Small language models are the future of agentic ai,” 2025. [Online]. Available: https://arxiv.org/abs/2506.02153

[42] R. Sharma and M. Mehta, “Small language models for agentic systems: A survey of architectures, capabilities, and deployment trade offs,” 2025. [Online]. Available: https://arxiv.org/abs/2510.03847

[43] J. Gou, B. Yu, S. J. Maybank, and D. Tao, “Knowledge distillation: A survey,” *International Journal of Computer Vision*, vol. 129, no. 6, p. 1789–1819, Mar. 2021. [Online]. Available: http://dx.doi.org/10.1007/s11263-021-01453-z

[44] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” 2015. [Online]. Available: https://arxiv.org/abs/1503.02531

[45] S. M. Bsharat, A. Myrzakhan, and Z. Shen, “Principled instructions are all you need for questioning llama-1/2, gpt-3.5/4,” 2024. [Online]. Available: https://arxiv.org/abs/2312.16171

[46] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[47] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[48] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[49] A. Roucher, A. V. del Moral, T. Wolf, L. von Werra, and E. Kaunismäki, “‘smolagents‘: a smol library to build great agentic systems.” https://github.com/

huggingface/smolagents, 2025.

[50] H. Chase, “LangChain,” Oct. 2022. [Online]. Available: https://github.com/langchain-ai/langchain

[51] J. Liu, “LlamaIndex,” 11 2022. [Online]. Available: https://github.com/jerryjliu/llama_index

[52] Y. Yu, L. Yao, Y. Xie, Q. Tan, J. Feng, Y. Li, and L. Wu, “Agentic memory: Learning unified long-term and short-term memory management for large language model agents,” 2026. [Online]. Available: https://arxiv.org/abs/2601.01885

[53] W.-C. Huang, W. Zhang, Y. Liang, Y. Bei, Y. Chen, T. Feng, X. Pan, Z. Tan, Y. Wang, T. Wei, S. Wu, R. Xu, L. Yang, R. Yang, W. Yang, C.-Y. Yeh, H. Zhang, H. Zhang, S. Zhu, H. P. Zou, W. Zhao, S. Wang, W. Xu, Z. Ke, Z. Hui, D. Li, Y. Wu, L. He, C. Wang, X. Xu, B. Huang, J. Tan, S. Heinecke, H. Wang, C. Xiong, A. A. Metwally, J. Yan, C.-Y. Lee, H. Zeng, Y. Xia, X. Wei, A. Payani, Y. Wang, H. Ma, W. Wang, C. Wang, Y. Zhang, X. Wang, Y. Zhang, J. You, H. Tong, X. Luo, X. Liu, Y. Sun, W. Wang, J. McAuley, J. Zou, J. Han, P. S. Yu, and K. Shu, “Rethinking memory mechanisms of foundation agents in the second half: A survey,” 2026. [Online]. Available: https://arxiv.org/abs/2602.06052

[54] Z. Wang, B. Yu, J. Zhao, W. Sun, S. Hou, S. Liang, X. Hu, Y. Han, and Y. Gan, “Karma: Augmenting embodied ai agents with long-and-short term memory systems,” 2025. [Online]. Available: https://arxiv.org/abs/2409.14908

[55] A. A. Ramanathan, “Architectural strategies for memory systems in agentic applications: Structured approaches to context management,” *Journal of International Crisis and Risk Communication Research*, vol. 9, no. 2, pp. 204–212, 2026, copyright - © 2026. This work is published under https://creativecommons.org/licenses/by/4.0/ (the “License”). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2026-04-02. [Online]. Available: https://www.proquest.com/scholarly-journals/architectural-strategies-memory-systems-agentic/docview/3309230001/se-2

[56] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez, “Memgpt: Towards llms as operating systems,” 2024. [Online]. Available: https://arxiv.org/abs/2310.08560

[57] Z. Liu, W. Yao, J. Zhang, L. Yang, Z. Liu, J. Tan, P. K. Choubey, T. Lan, J. Wu, H. Wang, S. Heinecke, C. Xiong, and S. Savarese, “Agentlite: A lightweight

library for building and advancing task-oriented llm agent system," 2024. [Online]. Available: https://arxiv.org/abs/2402.15538

[58] W. Xu, Z. Liang, K. Mei, H. Gao, J. Tan, and Y. Zhang, "A-mem: Agentic memory for llm agents," 2025. [Online]. Available: https://arxiv.org/abs/2502.12110

[59] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," 2023. [Online]. Available: https://arxiv.org/abs/2310.14021

[60] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," 2018. [Online]. Available: https://arxiv.org/abs/1603.09320

[61] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," 2017. [Online]. Available: https://arxiv.org/abs/1702.08734

[62] Pinecone Systems, Inc., "Pinecone: Vector database for machine learning applications," https://www.pinecone.io/, 2025, accessed: 2026-05-04.

[63] Qdrant Team, "Qdrant: Vector similarity search engine and database," https://qdrant.tech/, 2025, accessed: 2026-05-04.

[64] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2614–2627. [Online]. Available: https://doi.org/10.1145/3448016.3457550

[65] Chroma and contributors, "Chromadb: Open-source embedding database for llm applications," https://github.com/chroma-core/chroma, 2022, accessed: 2026-04-13.

[66] V. Sudhi, S. R. Bhat, M. Rudat, and R. Teucher, "Rag-ex: A generic framework for explaining retrieval augmented generation," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2776–2780. [Online]. Available: https://doi.org/10.1145/3626772.3657660

[67] S. Es, J. James, L. Espinosa-Anke, and S. Schockaert, "Ragas: Automated evaluation of retrieval augmented generation," 2025. [Online]. Available: https://arxiv.org/abs/2309.15217

[68] A. Salemi and H. Zamani, “Evaluating retrieval quality in retrieval-augmented generation,” 2024. [Online]. Available: https://arxiv.org/abs/2404.13781

[69] H. Han, Y. Wang, H. Shomer, K. Guo, J. Ding, Y. Lei, M. Halappanavar, R. A. Rossi, S. Mukherjee, X. Tang, Q. He, Z. Hua, B. Long, T. Zhao, N. Shah, A. Javari, Y. Xia, and J. Tang, “Retrieval-augmented generation with graphs (graphrag),” 2025. [Online]. Available: https://arxiv.org/abs/2501.00309

[70] Z. Guo, L. Xia, Y. Yu, T. Ao, and C. Huang, “Lightrag: Simple and fast retrieval-augmented generation,” 2025. [Online]. Available: https://arxiv.org/abs/2410.05779

[71] P. Liu, X. Liu, R. Yao, J. Liu, S. Meng, D. Wang, and J. Ma, “Hm-rag: Hierarchical multi-agent multimodal retrieval augmented generation,” *Proceedings of the 33rd ACM International Conference on Multimedia*, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:277857075

[72] I. Trummer, “Db-bert: a database tuning tool that ”reads the manual”,” 2021. [Online]. Available: https://arxiv.org/abs/2112.10925

[73] M. Khalefa, “Adaptive benchmarking for data system using llms,” in *2024 IEEE International Conference on Big Data (BigData)*, 2024, pp. 8703–8703.

[74] Streamlit Inc., “Streamlit: A faster way to build and share data apps,” https://streamlit.io/, 2026, accessed: 2026-05-04.

[75] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” *Transactions of the association for computational linguistics*, vol. 12, pp. 157–173, 2024.

[76] wolverdude and contributors, “Genson: A powerful, user-friendly json schema generator for python,” https://github.com/wolverdude/GenSON, 2025, version 1.3.0, accessed: 2026-05-04.

[77] S. Colvin and other contributors, “Pydantic: Data validation and settings management using python type annotations,” https://github.com/pydantic/pydantic, 2025, version 2.12.4, accessed: 2026-05-04.

[78] X. Fang, W. Xu, F. A. Tan, J. Zhang, Z. Hu, Y. Qi, S. Nickleach, D. Socolinsky, S. Sengamedu, and C. Faloutsos, “Large language models(llms) on tabular data: Prediction, generation, and understanding – a survey,” 2024. [Online]. Available: https://arxiv.org/abs/2402.17944

[79] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," 2020. [Online]. Available: https://arxiv.org/abs/2002.10957

[80] "all-minilm-l6-v2." [Online]. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[81] Ollama, "Ollama: Run large language models locally," https://github.com/ollama/ollama, 2023, accessed: 2026-04-14.

[82] M. H. Daniel Han and U. team, "Unsloth," 2023. [Online]. Available: https://github.com/unslothai/unsloth

[83] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Brew *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 38–45.

[84] "Qwen3," 2025.

[85] IBM Research, "Granite 4.0 language models," https://github.com/ibm-granite/granite-4.0-language-models, 2025, accessed: 2025-10-01.

[86] A. H. Liu, K. Khandelwal, S. Subramanian, V. Jouault, A. Rastogi, A. Sadé, A. Jeffares, A. Jiang, A. Cahill, A. Gavaudan, A. Sablayrolles, A. Héliou, A. You, A. Ehrenberg, A. Lo, A. Eliseev, A. Calvi, A. Sooriyarachchi, B. Bout, B. Rozière, B. D. Monicault, C. Lanfranchi, C. Barreau, C. Courtot, D. Grattarola, D. Dabert, D. de las Casas, E. Chane-Sane, F. Ahmed, G. Berrada, G. Ecrepont, G. Guinet, G. Novikov, G. Kunsch, G. Lample, G. Martin, G. Gupta, J. Ludziejewski, J. Rute, J. Studnia, J. Amar, J. Delas, J. S. Roberts, K. Yadav, K. Chandu, K. Jain, L. Aitchison, L. Fainsin, L. Blier, L. Zhao, L. Martin, L. Saulnier, L. Gao, M. Buyl, M. Jennings, M. Pellat, M. Prins, M. Poirée, M. Guillaumin, M. Dinot, M. Futeral, M. Darrin, M. Augustin, M. Chiquier, M. Schimpf, N. Grinsztajn, N. Gupta, N. Raghuraman, O. Bousquet, O. Duchenne, P. Wang, P. von Platen, P. Jacob, P. Wambergue, P. Kurylowicz, P. R. Muddireddy, P. Chagniot, P. Stock, P. Agrawal, Q. Torroba, R. Sauvestre, R. Soletskyi, R. Menneer, S. Vaze, S. Barry, S. Gandhi, S. Waghjale, S. Gandhi, S. Ghosh, S. Mishra, S. Aithal, S. Antoniak, T. L. Scao, T. Cachet, T. S. Sorg, T. Lavril, T. N. Saada, T. Chabal, T. Foubert, T. Robert, T. Wang, T. Lawson, T. Bewley, T. Bewley, T. Edwards, U. Jamil, U. Tomasini, V. Nemychnikova, V. Phung, V. Maladière, V. Richard, W. Bouaziz, W.-D. Li,

W. Marshall, X. Li, X. Yang, Y. E. Ouahidi, Y. Wang, Y. Tang, and Z. Ramzi, “Ministral 3,” 2026. [Online]. Available: https://arxiv.org/abs/2601.08584

APPENDICES

APPENDIX I

Prompts

Template-Based with Constrained decoding prompt (inside generation function - second inference call)

```
Given the user's request and documentation generate the necessary yaml configuraiton,
following the following format, without explanations or extra text.


params: {
    "experiments": [
        {
            "workloads": [
                {
                    "workload": "<workload_name>",
                    # One of: mdb, marketing-campaign-storm, marketing-campaign-flink,
                    # marketing-campaign-spark, ml-o, ml-n, ml-t-f,
                    # ml-o-str, ml-n-str, ml-t-f-str, sio, scpu, svm, i-p3

                    "device": "<device_name>"
                    # One of: xavier, nc8, orin, manager, jetson1, raspberrypi5
                }
                # Up to 3 workloads per experiment (they run simultaneously on the same node)
            ]
        }
    ]
}
```

Template-Based with Constrained decoding prompt (inside modification function - second inference call)

```
You are modifying an experiment YAML configuration.
Given the user's request and documentation, return ONLY a JSON array
of updates - no explanations or extra text.

Follow exactly the structure of the given YAML, and replace the leaf
values with the ones that are needed, if any changes are necessary.

User request: {query}
```

Table I.1: Template-Based with Constrained Decoding Prompts used for YAML experiment generation and modification.

Template-Based prompt (generation function definition)

```
Generates a YAML experiment configuration for one or more experiments.

Args:
    params: A class ExperimentList defining the experiments to run,
    which has the following structure:
        {
            "experiments": [
                {
                    "workloads": [
                        {
                            "workload": "<workload_name>",
                            # One of: mdb, marketing-campaign-storm, marketing-campaign-flink,
                            # marketing-campaign-spark, ml-o, ml-n, ml-t-f,
                            # ml-o-str, ml-n-str, ml-t-f-str, sio, scpu, svm, i-p3

                            "device": "<device_name>"
                            # One of: xavier, nc8, orin, manager, jetson1, raspberrypi5
                        }
                        # Up to 3 workloads per experiment (they run simultaneously on the same node)
                    ]
                }
            ]
        }

Returns:
    Success message with the generated structure,
    or a validation error.
```

Template-Based prompt (modification function definition)

```
Updates specific fields in an existing YAML experiment configuration.
Use this instead of regenerating the whole file when only a few values
need to change.

Args:
    params: A dict mapping Python subscript paths to their new values.
    For example:
        {
            config["experiments"][0]["experiment"]["record_name"]: "new-name",    # < 20 chars

            config["experiments"][1]["experiment"]["duration"]: "20m",

            config["experiments"][0]["experiment"]["workloads"][0]
            ["parameters"]["model_volume_path"]: "/data/models",

            config["experiments"][0]["experiment"]["workloads"][1]
            ["parameters"]["options"][0]["--io"]: 4
        }

    All indices are zero-based.

Returns:
    Success message listing updated fields,
    or a validation error.
```

Table I.2: Template-Based Prompts used for YAML experiment generation and modification.