Agentic Placement of Microservices on the Computing Continuum

Kunal Rao, Giuseppe Coviello and Srimat Chakradhar
NEC Laboratories America, Inc.
Princeton, NJ, USA
email:{kunal, giuseppe.coviello, chak}@nec-labs.com

***Abstract*—Deploying microservices across the computing continuum (edge–cloud) requires placement decisions that adapt to workload variation and heterogeneous infrastructure, yet existing solutions often rely on static policies or opaque heuristics. We present *Bellona* a system for *reliable and auditable Large Language Model (LLM)-driven workflow execution* that combines a declarative specification language with a runtime that orchestrates tool calls, conditional control flow, and structured LLM reasoning. Using *Bellona*, we implement an agentic placement workflow that automatically recommends edge or cloud execution. The workflow uses structured prompts and verifiable tool interactions to (i) parse placement and latency-report instructions, (ii) update the latency log, and (iii) select placements based on measured latency improvement thresholds. We evaluate the resulting agent on two representative microservices-based video analytics applications (human-attributes detection and face recognition) over two days of varying workload. Across 1,440 placement decisions per service, the agent achieves accuracies of 94.66%/84.94% (human-attributes detection, Day1/Day2) and 80.91%/96.53% (face recognition, Day1/Day2) with GPT-4o; with GPT-5, accuracy increases to 98.82%/99.45% (human-attributes detection) and 99.31%/99.8% (face recognition). These results demonstrate that *Bellona* can support practical, self-improving agentic control for placement of microservices on the computing continuum.**

***Keywords*-Computing continuum; Edge–cloud placement; Microservices; Agentic workflows; Large language models; Workflow orchestration; Latency-aware decision making.**

I. INTRODUCTION

The computing continuum spans resource-constrained edge nodes and elastic cloud infrastructure, enabling latency-sensitive applications while accommodating workload bursts and complex analytics. A persistent challenge in this setting is *microservice placement*: deciding, for each workload instance, whether a service should execute at the edge or in the cloud. Placement policies must contend with heterogeneous hardware, time-varying contention, and application-dependent trade-offs, and they are often deployed as static rules or hand-tuned heuristics that are brittle under changing conditions.

Recent advances in Large Language Models (LLMs) make it tempting to build "agentic" controllers that interpret operator intents, query measurements, and take actions (e.g., selecting a placement or updating state) through tool calls. In practice, however, LLM-based controllers can be difficult to operationalize: the control logic is frequently embedded in ad-hoc prompt chains, intermediate state is implicit, and the resulting behavior is hard to reproduce, debug, and audit. Existing agent frameworks (e.g., LangChain [1], Semantic Kernel [2], and DSPy [3]) simplify tool calling and composition, but they do not provide a principled substrate for specifying multi-step agent behavior with explicit state, control flow, and verifiable execution semantics.

We present *Bellona*, a system for *reliable and auditable execution of agentic workflows*. *Bellona* couples (i) a declarative, human-readable specification language for defining tasks, data dependencies, and control flow and (ii) a runtime that executes these workflows with standardized tool integration and structured LLM interactions. The specification layer supports conditional branches, iterative and parallel tasks, reusable procedures, and template-based variable substitution, while the runtime manages context, orchestrates tool calls, and enforces deterministic control flow around inherently stochastic model outputs.

To demonstrate *Bellona* on a concrete continuum-management task, we implement an agentic placement workflow that automatically recommends edge or cloud execution and incrementally learns from measurements. The workflow accepts natural-language instructions of two types: (1) requests to recommend placement for a given workload and (2) reports of observed latency for a given workload–infrastructure pair. It uses tool calls to maintain a shared Comma-Separated Values (CSV) latency log and employs structured prompts to (i) extract workload/infrastructure/latency entities, (ii) avoid duplicate records, and (iii) recommend the next placement based on the current coverage of measurements and a latency-improvement threshold once both edge and cloud observations are available.

We evaluate this agent on two representative video-analytics microservices (human-attributes detection and face recognition) under workload variation across two days and observe up to $\sim$99.8% accuracy, demonstrating that *Bellona* can support practical agentic control for placement on the computing continuum.

The main contributions of this paper are as follows:

- We present *Bellona*, a specification-first system that combines declarative workflow definitions with a runtime for reliable and auditable LLM+tool workflow execution.
- We use *Bellona* to implement an agentic placement workflow for the edge–cloud continuum that interprets natural-language instructions, maintains explicit state in a latency log, and applies a measurement-driven placement policy.
- We evaluate the resulting workflow on two video-analytics microservices under a real workload trace and show that it achieves high placement accuracy across 1,440 decisions per day.

The remainder of the paper is structured as follows: Section II discusses related work on continuum placement and agentic orchestration; Section III describes *Bellona*, including the specification language and runtime architecture; Section IV

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6

presents the placement workflow implemented in *Bellona*; Section V reports experimental results and Section VI concludes.

II. Related Work

Prior work on microservice placement across the edge–cloud computing continuum spans (i) rule-based policies (e.g., fixed latency thresholds), (ii) optimization-based schedulers, and (iii) learning-based controllers that map observed load, latency, and resource availability to placement actions. In practice, these solutions are embedded in orchestration and monitoring stacks that continuously collect telemetry and trigger (re)deployment. Recent work has also begun exploring how LLMs can assist continuum-management logic, e.g., ECO-LLM [4], as well as complementary efforts on edge–cloud LLM systems and collaborative inference [5]–[7]. While effective, operationalizing these approaches often requires substantial "glue" code to connect operator intent, measurement pipelines, and application-specific objectives into repeatable decision procedures.

From a systems perspective, real-world placement is naturally a multi-step workflow: gathering measurements, updating state, evaluating a policy, and actuating changes. General-purpose workflow orchestrators, such as Apache Airflow [8], Prefect [9], Dagster [10], Temporal [11], and managed services like AWS Step Functions [12] and Azure Durable Functions [13] provide explicit control flow, retries, and scheduling for such pipelines; DagOn* [14] similarly targets DAG execution across heterogeneous resources. However, these systems largely assume deterministic task logic and do not treat LLM calls, tool-driven reasoning, or output verification as first-class components, making it difficult to build a reliable, auditable LLM-in-the-loop control loop.

LLM agent frameworks address tool use and intent interpretation by allowing models to decide which tools to call and how to compose actions. Surveys on agentic AI [15], [16] and multi-agent systems [17] describe common architectures, while systems, such as AutoGen [18], CAMEL [19], HuggingGPT [20], and CrewAI [21] demonstrate coordination across multiple LLM roles and external tools. Tool-use methods including Chain-of-Thought [22], ReAct [23], MRKL [24], Toolformer [25], and PAL [26] show how reasoning can be interleaved with actions. In many implementations, however, control flow and state transitions remain implicit in prompt chains, complicating reproducibility, debugging, and governance.

Finally, low-code automation platforms (e.g., Dify [27], n8n [28], and Windmill [29]) and LLM programming systems such as DSPy [3] and XPF [30] make it easier to compose LLM-centric pipelines. *Bellona* complements these efforts by providing a specification-first workflow substrate that (i) makes state and control flow explicit, (ii) treats LLM calls and tool calls as first-class tasks, and (iii) supports reliable, auditable execution. These properties are particularly important for operational placement workflows on the computing continuum.

III. System Design and Implementation

Bellona provides an end-to-end architecture for defining and executing *agentic workflows* in which LLM calls, tool calls, and control flow are explicit, repeatable, and auditable. Figure 1 shows the two main layers of the system: a *Specification Layer* that describes workflow structure and state dependencies, and a *Runtime Layer* that enforces deterministic control flow around inherently stochastic model outputs, orchestrates tool invocation, and records execution traces.

A. Workflow specification

Bellona workflows are written in a declarative YAML format with (i) metadata and typed inputs, (ii) labeled tasks (LLM calls, tool calls, and control-flow constructs such as conditionals/loops/parallel blocks), and (iii) reusable procedures. Jinja2-style templates (e.g., `{{instruction_type}}`) connect task outputs to subsequent inputs, making dataflow and state transitions explicit for versioning and auditability. In addition to hand-authoring YAML, *Bellona* includes a lightweight *workflow generator* that can translate a natural-language instruction into an initial workflow draft, which we then review and (if needed) minimally edit before execution.

B. Runtime execution

The runtime parses and validates the workflow, then executes tasks while maintaining a scoped *context stack* for variables across nested procedure calls. Tool invocations are mediated through a unified gateway (via MCP), which standardizes heterogeneous tools behind a common schema. For LLM tasks, the runtime expands templates, executes the model with optional tool-use loops, validates outputs when rubrics are present, and records structured traces (inputs, outputs, timing, and tool results). Parallel blocks are executed concurrently but aggregated deterministically.

Together, the specification and runtime layers turn prompt chains into an explicit program with traceable state, control flow, and tool interactions. We next describe the runtime execution engine in more detail.

C. Runtime Execution Engine

This section presents the execution semantics of *Bellona*'s runtime, which is implemented as a structured dispatcher around a context stack, a rubric-aware retry loop for LLM tasks, and a unified gateway for tool invocation. The executor coordinates templates, control flow, quality validation, caching, and observability to provide reliable and auditable execution.

1) Execution Model and Context: The runtime exposes a single entry point `Run(args, mcpCfg)` that validates the workflow, assigns task identifiers, initializes the Model Context Protocol (MCP) toolkit, and then invokes the root procedure. Each procedure call pushes a new frame on a *context stack* that stores variables, including the latest task output for dataflow chaining. Context updates use `addToContext(key, value)`, while `pushContext`/`popContext` delimit scope. Templates are

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

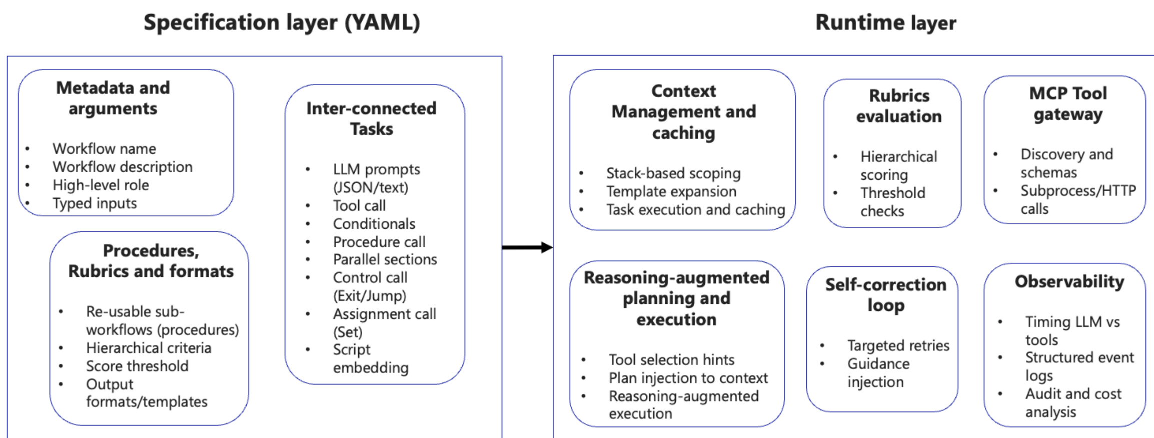


Figure 1. Overall architecture of *Bellona*. The specification layer defines YAML-based workflows (metadata, typed arguments, tasks, and reusable procedures) with optional rubrics for output validation. The runtime layer orchestrates execution using a scoped context, MCP-based tool invocation, structured LLM interactions, and trace logging for auditability.

expanded with a strict Jinja2-compatible engine before each task or tool call, which guarantees early surfacing of missing variables.

2) Task Dispatch and Control Flow: The executor visits tasks in program order and dispatches by type (LLM, tool call, conditional, procedure call, and simple state updates such as `set`). Control primitives such as `exit` and `jump` provide early termination and explicit error-recovery paths, while conditionals can be evaluated either by the LLM (for natural-language predicates) or deterministically (e.g., via a Python expression) when reproducibility is critical.

3) LLM Orchestration, Tools, and Validation: For each LLM task, the runtime expands templates, executes the model, and (when enabled) iteratively services any emitted tool calls through the MCP toolkit until the model returns a final response. When a task declares `withJSONOutput`, the runtime parses and type-checks the output before committing it to the context. If a rubric is provided, the runtime evaluates the output and may trigger a small number of guided retries using feedback derived from failed rubric criteria.

4) Caching and observability: Bellona supports quality-aware caching for LLM tasks: cached results are reused only when they correspond to the fully expanded prompt and satisfy any associated rubric. During execution, the runtime emits structured traces per task (prompt, tool calls, outputs, timing), which enables post-hoc auditing and debugging.

5) Determinism aids: To improve reproducibility, template expansion runs in strict mode to surface missing variables early, tool names are validated against the active MCP registry, and conditionals can be evaluated deterministically (e.g., via a Python expression) when required.

The *Bellona* runtime layer thus couples a scoped context stack, an optional rubric-driven retry loop, and an MCP-based tool gateway with explicit control primitives such as `exit` and `jump`. This combination provides reliable progress, verifiable outputs, and rich observability for LLM+tool workflows while preserving modularity through reusable procedures and typed arguments.

IV. AGENTIC WORKFLOW FOR PLACEMENT

We first provide a natural-language "programming" instruction to *Bellona*'s workflow generator. The generator turns this instruction into an executable workflow that (i) interprets operator messages, (ii) maintains explicit state in a latency log, and (iii) returns an edge–cloud placement recommendation.

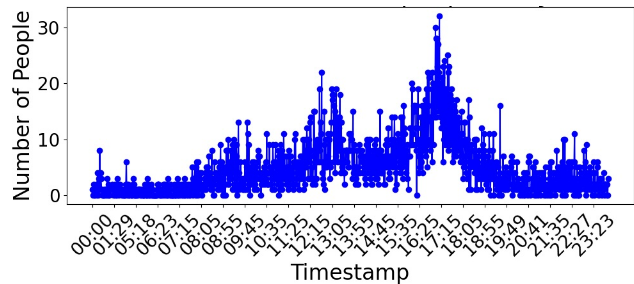
```
   Create a placement orchestrator that
responds with a JSON object {"placement":
"edge" or "cloud"}.    Maintain a CSV
file latency_report.csv with rows
<workload>,<infrastructure>,<latency>.
   The orchestrator receives two instruction
types:
(1) Recommend placement for workload:
<workload> and
(2) Latency on <infrastructure> for
workload: <workload> is <latency>.
   Policy: If a workload has no entries,
recommend edge. If exactly one of edge/cloud
is recorded, recommend the missing one to
collect the other measurement. If both are
recorded, recommend cloud only when it is
at least 20% faster than edge; otherwise
recommend edge.
```

The full workflow specification used in our experiments (generated by *Bellona*'s workflow generator and then lightly edited to ensure it executes correctly) is included in Appendix A.

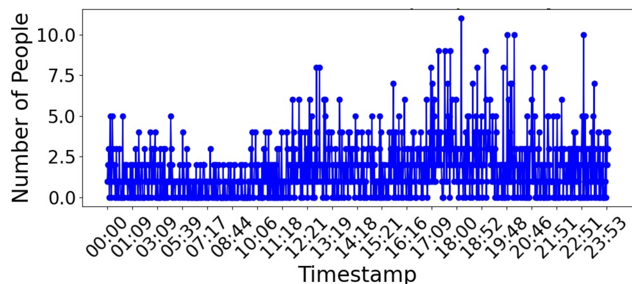
The workflow proceeds in four stages. It first classifies the incoming message as either a placement request or a latency report (Task `instruction_type`). For a latency report, it extracts `workload`, `infrastructure`, and `latency`, normalizes them into a CSV row, checks the existing log for duplicates, and appends the new measurement if needed. For a placement request, it reads the current log, extracts the workload identifier, and applies the policy in the prompt: cold-start on `edge`, then explore the missing infrastructure, and finally exploit `cloud` only when it exceeds the 20% improvement threshold. All state changes (file reads/appends)

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6



(a). People (Day 1)



(b). People (Day 2)

Figure 2. Variation in workload

occur through explicit tool calls, making the control loop reproducible and auditable.

V. Experimental results

A. Workload trace

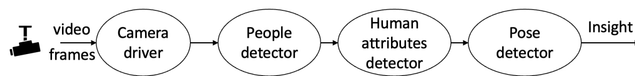
We drive the placement workflow using a real-world public trace from the City of Melbourne pedestrian counting system [31]. The trace reports the number of people observed at each sensor location over time. We select one representative location and use two 24-hour periods (one weekday and one weekend day). Figure 2 visualizes the resulting workload variation, which we treat as the per-minute input rate for both applications.

B. Applications and deployment options

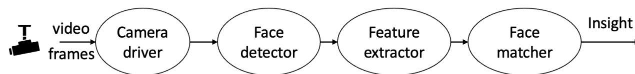
Figure 3 shows the two representative microservice-based video-analytics pipelines we study: Human-Attributes Detection (HAD) and Face Recognition (FR). We consider two execution tiers: *edge* and *cloud*. Our edge tier comprises AMD Ryzen 9 5900HX, 8 cores machine, while the cloud tier uses an AWS `c4.8xlarge` VM instance. In both pipelines, the upstream microservices (`camera-driver`, `people detector`, and `face detector`) always run on the edge, while the downstream microservices (`human attributes detector`, `pose detector`, `feature extractor`, and `face matcher`) are the ones placed either on edge or on cloud.

C. Latency characterization and baseline

We first characterize end-to-end pipeline latency under different workloads for each placement option. In Figure 4, *Edge* denotes placement where all microservices execute on the edge machine. *Cloud* denotes placement where

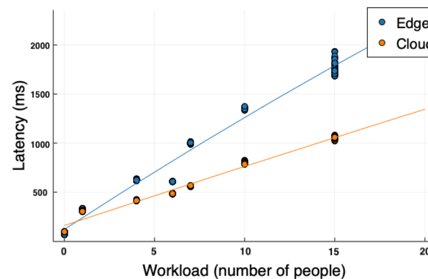


(a). Human attributes detection pipeline

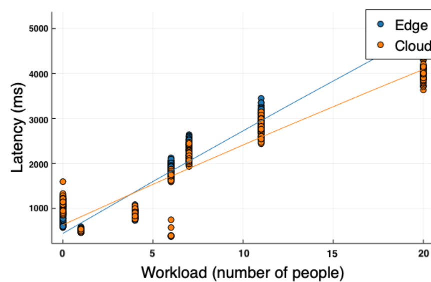


(b). Face recognition pipeline

Figure 3. Representative video analytics application pipelines



(a). Human Attributes Detection



(b). Face Recognition

Figure 4. Characterization of application latency

the downstream stages execute on the cloud VM (HAD: `human attributes detector` and `pose detector`; FR: `feature extractor` and `face matcher`). We run only one pipeline/placement at a time (i.e., no concurrent pipelines).

Using these measurements, we define an *oracle baseline* that mirrors the workflow policy: for each workload value, it recommends `cloud` only if the characterized cloud latency is at least 20% lower than the characterized edge latency; otherwise it recommends `edge`. We use this baseline as the reference policy when computing workflow-driven placement decision accuracy.

D. Agentic Workflow Placement accuracy

Table I reports how often the agentic workflow matches the oracle baseline over the two days. We evaluate two LLM backends (GPT-4o and GPT-5), and we make one placement decision per minute, yielding 1,440 decisions per day (missing trace values are forward-filled with the most recent observation, and workflow is pre-warmed with latency numbers recorded for edge and cloud for different workloads).

We report (i) total decisions (TPD), (ii) incorrect `edge` recommendations when the baseline is `cloud` (TIP Edge), (iii) incorrect `cloud` recommendations when the baseline is `edge` (TIP Cloud), and the resulting accuracy. With GPT-4o,

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6

Table I. PLACEMENT ACCURACY OF THE AGENTIC WORKFLOW AGAINST THE ORACLE BASELINE (HAD: HUMAN ATTRIBUTES DETECTOR; FR: FACE RECOGNITION; TPD: TOTAL PLACEMENT DECISIONS; TIP: TOTAL INCORRECT PLACEMENTS).

	HAD				**FR**			
TPD	1440				1440			
	GPT-4o		GPT-5		GPT-4o		GPT-5	
	Day1	**Day2**	**Day1**	**Day2**	**Day1**	**Day2**	**Day1**	**Day2**
TIP Edge (E)	42	0	16	8	0	0	0	0
TIP Cloud (C)	35	217	1	0	275	50	10	3
TIP (E+C)	77	217	17	8	275	50	10	3
Error (%)	5.34	15.06	1.18	0.55	19.09	3.47	0.69	0.20
Accuracy (%)	**94.66**	**84.94**	**98.82**	**99.45**	**80.91**	**96.53**	**99.31**	**99.8**

the workflow achieves accuracies of 94.66%/84.94% for HAD (Day1/Day2) and 80.91%/96.53% for FR (Day1/Day2). With GPT-5, accuracy increases to 98.82%/99.45% for HAD and 99.31%/99.8% for FR.

VI. Conclusion and Future Work

In this paper, we presented *Bellona*, a specification-first system for executing agentic workflows with explicit state, conditional control flow, and verifiable tool calls, and used it to implement an LLM-driven placement workflow that (i) interprets natural-language placement and latency-report instructions, (ii) maintains a persistent CSV latency log, and (iii) recommends `edge` or `cloud` placement using a measurement-driven policy. Across two video-analytics applications and two days of workload variation, the agentic workflow achieves accuracies of 94.66%/84.94% (HAD, Day1/Day2) and 80.91%/96.53% (FR, Day1/Day2) with GPT-4o; with GPT-5, accuracy increases to 98.82%/99.45% (HAD) and 99.31%/99.8% (FR), suggesting that declarative, tool-grounded LLM workflows can provide practical continuum control while remaining reproducible and auditable. Although we demonstrate with a specific class of applications and using only 2 tiers of computing, future work can investigate the impact with different LLM models on multiple tiers of computing across a diverse set of applications.

Appendix A
Placement Orchestrator Agentic Workflow

This appendix contains the full *Bellona* workflow specification for the placement orchestrator used in our experiments.

```
title: Placement Orchestrator
description: Orchestrates placement on edge or cloud
arguments:
  instruction:
    type: string
tasks:
  - label: instruction_type
    llm:
      prompt: |
        From the instruction below, return a compact JSON
  object with the following key:
        - type: the type of the instruction, it can be
  either recommend_placement or record_latency.

        The instruction template for type
  "recommend_placement" will be like this:

        "Recommend placement for workload: <workload>"

        The instruction template for type "record_latency"
will be like this:

        "Latency on <infrastructure> for workload:
<workload> is <latency>"

        Instruction:
        {{ instruction }}
      withJSONOutput: true
      withoutTools: true
- label: process_instruction
  conditional:
    cases:
      - condition: "{{instruction_type.type | tojson }}
== 'record_latency'"
        tasks:
          - label: extract_reported_latency
            llm:
              prompt: |
                From the instruction below, return a
JSON object with the following keys:
                - infrastructure: reported
infrastructure (either cloud or edge)
                - workload: reported workload number
                - latency: reported latency

                The reported instruction template will
be like this:

                "Latency on <infrastructure> for
workload: <workload> is <latency>"

                Instruction:
                {{ instruction }}
              withJSONOutput: true
              withoutTools: true
          - label: latency_csv
            llm:
              prompt: |
                From the JSON below, extract and
generate a comma separated string in this format:
                <workload>,<infrastructure>,<latency>

                Keep the workload as an integer.

                JSON:
                {{ extract_reported_latency}}

                ONLY output the comma separated string.
NOTHING ELSE.
          - label: review_prior_reported_latency
            tool_call:
              name: "readFile"
              arguments:
                filename: "latency_report.csv"
          - label: check_if_record_exists
            llm:
              prompt: |
                Below is a latency report in this format:
                <workload>,<infrastructure>,<latency>

                Reply "yes" if the specified <workload>
and <infrastructure> combination already exists in the
prior data.
                Reply "no" if it does not exist in prior
data.

                Each row in prior data has the same
format i.e. <workload>,<infrastructure>,<latency>

                Reply only "yes" or "no". NOTHING ELSE.

                latency report:
                {{ latency_csv }}

                Prior data:
                {{ review_prior_reported_latency }}

          - label: record_latency
            conditional:
              cases:
                - condition: '{{check_if_record_exists}}
== "no"'
                  tasks:
```

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6

```
                    - label: append_reported_latency
                      tool_call:
                        name: "appendToFile"
                        arguments:
                          filename: "latency_report.csv"
                          text: "{{ latency_csv }}"
                    - label: respond
                      llm:
                        prompt: |
                          Respond to the user saying
that the specified latency is recorded.
              default:
                - label: respond
                  llm:
                    prompt: |
                      Respond to the user saying that
the latency for specified workload and infrastructure
combination is already recorded.

      - condition: "{{instruction_type.type | tojson }}
== 'recommend_placement'"
        tasks:
          - label: review_prior_reported_latency
            tool_call:
              name: "readFile"
              arguments:
                filename: "latency_report.csv"
          - label: extract_workload
            llm:
              prompt: |
                Extract the workload number from the
below instruction:

                Instruction:
                {{ instruction }}

                Just give the workload number. NOTHING
ELSE.

          - label: recommend_placement
            llm:
              prompt: |
                You recommend placement on edge or cloud
in JSON format. The JSON object has a single key:
                - placement: value is either "edge" or
"cloud".

                Based on the below latency data, see if
data for workload: {{extract_workload}} exists.
                DO NOT make up any data. Remember that
latency data can be empty initially.

                If it does not exist, return JSON object
with placement as "edge".

                If only one of "edge" or "cloud" data
exists, then return JSON object with placement as the
MISSING one among "edge" or "cloud".
                For example, if data for "edge" exists,
then return "cloud" and vice versa.

                If both are present, return JSON object
with placement as "cloud" ONLY IF latency improvement
over edge is greater than 20%,
                otherwise return JSON object with
placement as "edge".

                Latency data:
                {{review_prior_reported_latency}}

              withJSONOutput: true
              withoutTools: true
    default:
      - label: unknown_instruction
        llm:
          prompt: |
            Say that the instruction is not recognized.
Instruction should either report latency or request
recommendation.
```

REFERENCES

[1] LangChain, *Langchain*, https://www.langchain.com/, Last accessed Feb. 25, 2026.

[2] Microsoft, *Semantic Kernel documentation*, [https://learn.microsoft.com/en-us/semantic-kernel/], Last accessed Feb. 25, 2026, 2025.

[3] O. Khattab *et al.*, "DSPy: Compiling declarative language model calls into self-improving pipelines", 2024.

[4] K. Rao *et al.*, "Eco-llm: Llm-based edge cloud optimization", in *Proceedings of the 2024 Workshop on AI For Systems*, ser. AI4Sys '24, Pisa, Italy: Association for Computing Machinery, 2024, pp. 7–12, ISBN: 9798400706523. DOI: 10.1145/3660605.3660941.

[5] Z. Zhang *et al.*, *Creating edge ai from cloud-based large language models*, arXiv preprint arXiv:2410.18080, 2024.

[6] J. Chen *et al.*, *Edgeshard: Efficient llm inference on edge devices via sharded kv cache*, arXiv preprint arXiv:2405.14371, 2024.

[7] J. Xu *et al.*, *Ce-collm: Computation-efficient collaborative large language model inference for edge-cloud environments*, arXiv preprint arXiv:2411.02829, 2024.

[8] *Apache airflow*, https://airflow.apache.org/, Last accessed Feb. 25, 2026.

[9] *Prefect documentation*, https://docs.prefect.io/latest/, Last accessed Feb. 25, 2026.

[10] *Dagster documentation*, https://docs.dagster.io/, Last accessed Feb. 25, 2026.

[11] *Temporal*, https://temporal.io/, Last accessed Feb. 25, 2026.

[12] *Aws step functions developer guide*, Last accessed Feb. 25, 2026, Amazon Web Services.

[13] *Azure durable functions overview*, Last accessed Feb. 25, 2026, Microsoft.

[14] R. Montella, D. Di Luccio, and S. Kosta, "Dagon*: Executing direct acyclic graphs as parallel jobs on anything", in *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, IEEE, 2018, pp. 64–73.

[15] D. B. Acharya, K. Kuppan, and B. Divya, "Agentic AI: Autonomous Intelligence for Complex Goals—A Comprehensive Survey", *IEEE Access*, vol. 13, pp. 18 912–18 936, 2025. DOI: 10.1109/ACCESS.2025.3532853.

[16] A. K. Pati, "Agentic AI: A Comprehensive Survey of Technologies, Applications, and Societal Implications", *IEEE Access*, vol. 13, pp. 151 824–151 837, 2025. DOI: 10.1109/ACCESS.2025.3585609.

[17] S. Han, Q. Zhang, Y. Yao, W. Jin, and Z. Xu, *Llm multi-agent systems: Challenges and open problems*, 2025. arXiv: 2402.03578 [cs.MA].

[18] *Autogen: Enabling next-gen llm applications via multi-agent conversation framework*, 2023. arXiv: 2308.08155.

[19] *Camel: Communicative agents for "mind" exploration*, 2023. arXiv: 2303.17760.

[20] *HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in HuggingFace*, 2023. arXiv: 2303.17580.

[21] *Crewai documentation*, https://docs.crewai.com/, Last accessed Feb. 25, 2026.

[22] J. Wei *et al.*, "Chain of thought prompting elicits reasoning in large language models", *Neural Inf Process Syst*, vol. abs/2201.11903, Jan. 2022.

[23] *React: Synergizing reasoning and acting in language models*, 2022. arXiv: 2210.03629.

[24] *Mrkl systems*, 2022. arXiv: 2205.00445.

[25] *Toolformer: Language models can teach themselves to use tools*, 2023. arXiv: 2302.04761.

[26] *Program-aided language models*, 2022. arXiv: 2211.10435.

[27] *Dify*, https://dify.ai/, Last accessed Feb. 25, 2026.

[28] *N8n*, https://n8n.io/, Last accessed Feb. 25, 2026.

[29] *Windmill*, https://www.windmill.dev/, Last accessed Feb. 25, 2026.

[30] K. Rao, G. Coviello, G. Mellone, C. G. De Vita, and S. Chakradhar, "XPF: Agentic AI System for Business Workflow Automation", in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '25, University of Notre Dame Conference Facilities, Notre Dame, IN, USA: Association for Computing Machinery, 2025, pp. 1–6, ISBN: 9798400718694. DOI: 10.1145/3731545.3743644.

[31] City of Melbourne, *Pedestrian Counting System — Past Hour (counts per minute)*, https://melbournetestbed.opendatasoft.com/explore/dataset/pedestrian-counting-system-past-hour-counts-per-minute/information/?sort=-location_id, Last accessed Feb. 25, 2026, 2022.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-373-6