Highlights

Intent Engine: Natural-Language Intent Translation for Intent-Driven Orchestration in the Compute Continuum

Koushikur Islam,Rodrigo N. Calheiros

- Translates natural-language service placement intents into orchestration-consumable Service-level Objective (SLO) artifacts.
- Provides retrieval-augmented and schema-bounded LLM-based intent-to-specification construction.
- Reduces the need for specialized distributed-systems expertise in expressing service placement goals.
- Supports adaptable integration with intent-driven placement algorithms and orchestration frameworks.

Intent Engine: Natural-Language Intent Translation for Intent-Driven Orchestration in the Compute Continuum

Koushikur Islam[a,*], Rodrigo N. Calheiros[a]

[a]Western Sydney University, Penrith, NSW, Australia

ARTICLE INFO

Keywords:
Intent-Driven Orchestration
Service Placement
Microservices
Service-level Objectives (SLOs)
Edge-Cloud Continuum
Large Language Model (LLM)

ABSTRACT

Microservice placement in the compute continuum is driven by low-level Service-level Objectives (SLOs), but requiring users to specify metric-level constraints creates an adoption barrier and increases misconfiguration risk. Although large language models (LLMs) can interpret natural-language intents, direct generation of orchestration-consumable SLO artifacts remains unreliable due to unsupported constraints, incorrect grounded values, and schema violations. These errors can propagate to downstream placement logic and produce infeasible or incorrect placements. This paper presents *Intent Engine*, a natural-language intent translation architecture that constructs validated SLO artifacts for compute-continuum service placement. *Intent Engine* acts as an intent acquisition and SLO construction layer for existing intent-driven orchestration and placement frameworks; it does not perform placement or runtime QoS optimization. The architecture combines schema-constrained extraction, retrieval-grounded value construction from monitored infrastructure state, and validation against supported constraints before emitting the final SLO artifact. We evaluate *Intent Engine* using a 716-record intent-to-SLO dataset derived from an edge–cloud testbed, including valid and invalid intents. Across GPT-4.1 mini, Claude Sonnet 4.5, and DeepSeek V4-Flash, *Intent Engine* outperforms prompting baselines and a non-LLM rule-based parser. With GPT-4.1 mini, it achieves 0.941 total F1 Score and reduces aggregate hallucination by 85.1%, while lowering downstream placement failure from 30.8% to 2.1%.

1. Introduction

Software applications have driven a paradigm shift toward *microservices*, enabling improved scalability, flexibility, resilience, and faster delivery. By decomposing software into independently deployable components, microservices support dynamic scaling, rapid rollout, and reduced latency [1]. Modern microservice-based systems primarily rely on cloud infrastructures to achieve scalability, cost efficiency, and availability; however, centralized data centers introduce network latency due to limited geographic proximity to users [2]. This limitation has led to the emergence of edge computing.

Edge computing [3] mitigates latency by processing data closer to its source in real time. Nevertheless, resource constraints at the edge restrict scalability, computational capacity, and availability compared to cloud environments. To balance these trade-offs, the *compute continuum* integrates edge and cloud resources, enabling microservice deployment across heterogeneous infrastructures.

The compute continuum spans endpoints, edge, and cloud layers, distributing services to reduce latency while improving throughput and scalability [4]. Within this environment, microservice placement becomes critical for meeting user requirements such as low latency, high availability, storage efficiency, and quality of service (QoS) [5]. Applications including live streaming, AI inference, and digital advertising are particularly sensitive to these placement decisions.

Service-level Objectives (SLOs) are commonly used to guide placement across geographically distributed resources [5, 6, 7]. However, low-level SLO specification imposes a significant barrier for users lacking system expertise. Misconfigured placements can lead to QoS violations, service disruptions, and unexpected costs. Given the complexity and dynamic nature of continuum environments, accurate placement decisions are therefore essential for reliable operation.

Intent-driven orchestration (IDO) addresses this challenge by allowing users to express high-level goals, or intents, that describe desired outcomes rather than specific configurations [8]. Inspired by Intent-Based Networking (IBN) [9, 10], IDO autonomously translates these objectives into operational actions, abstracting low-level system details and reducing administrative burden. This abstraction lowers the expertise barrier and promotes scalable orchestration across heterogeneous resources.

Despite these benefits, existing orchestration frameworks [1, 11, 12, 13, 14, 15] typically accept intents as structured SLO specifications rather than unstructured natural language. Users must still provide precise metrics and system parameters, requiring domain knowledge and increasing the risk of misconfiguration, particularly under dynamic and heterogeneous conditions.

Recent advances in large pre-trained language models (PLMs) and large language models (LLMs) have demonstrated strong capabilities in natural language understanding and reasoning [16, 17]. These capabilities suggest that LLMs could translate human-readable intents into formal SLO

*Corresponding author
k.shohag@westernsydney.edu.au (K. Islam); r.calheiros@westernsydney.edu.au (R.N. Calheiros)
ORCID(s): 0009-0003-4619-7873 (K. Islam); 0000-0001-7435-2445 (R.N. Calheiros)

specifications consumable by orchestration frameworks, thereby simplifying service placement for operators and DevOps practitioners.

However, employing LLMs to generate system-critical specifications introduces new risks. LLMs are prone to hallucinations, producing outputs that deviate from user intent or contextual correctness [18]. In placement decisions, such errors can result in invalid configurations or system failures. Because continuum resources fluctuate dynamically, purely generative translations lack the guarantees required for reliable control.

In our prior work, *MicroIntent* [19], we found that end-to-end LLM-based translation of natural-language placement intents can produce hallucinated or incorrect SLO constraints, even when supplied with contextual information. This is problematic in the compute continuum, where fluctuating resource states require reliable and grounded control-plane specifications rather than prompt-only generation. These findings indicate that relying solely on end-to-end LLM inference can lead to misconfigurations, user intent violations, and unanticipated system impacts.

Based on these insights, we argue that natural-language intent translation is a control-plane specification-construction problem that is different from typical generative language tasks. Current IDO frameworks require structured SLO inputs, whereas users often express placement intents as incomplete, ambiguous, or context-dependent details. A reliable acquisition layer must parse the intent, ground context-dependent requirements in monitored infrastructure state, check the result against the supported schema, and emit only validated SLO artifacts for downstream orchestration.

In this paper, we propose *Intent Engine*, a natural-language intent translation architecture that converts service placement intents into platform-agnostic SLO specifications consumable by existing IDO and placement frameworks [1, 11, 12, 13, 14, 15]. *Intent Engine* acts as an intent acquisition and SLO construction layer: it produces validated SLO artifacts to drive accurate intent-driven downstream orchestration.

Since inaccurate translations can directly lead to infeasible or incorrect placements, our evaluation measures translation correctness, grounding accuracy, hallucination reduction, invalid-intent rejection, latency, scalability, and downstream placement impact. This paper makes the following contributions:

- We introduce *Intent Engine*, an intent acquisition system that converts natural-language service placement intents into validated, orchestration-consumable SLO artifacts.
- We design a multi-stage SLO construction pipeline that separates semantic extraction, intermediate SLO representation, infrastructure-aware grounding, and schema validation.
- We introduce infrastructure-aware grounding to resolve implicit constraints, such as highest or lowest resource values, from monitored compute-continuum state.
- We evaluate *Intent Engine* using constraint-level F1 Score, Exact Match, Jaccard similarity, hallucination rate, invalid-intent rejection, context retrieval ablation, latency, scalability, and downstream placement-failure impact.
- We release a 716-record intent-to-SLO dataset from a real compute-continuum testbed, covering valid, ambiguous, conflicting, malformed, and unsupported intents.[1]

Due to the language translation focus of *Intent Engine*, tasks such as placement, deployment, or runtime QoS optimization are outside the scope of this paper.

The rest of the paper is organized as follows. Section 2 reviews existing work. Section 3 presents the *Intent Engine* architecture. Section 4 describes the implementation and compute-continuum testbed. Section 5 evaluates SLO construction accuracy, grounding effectiveness, hallucination reduction, failure handling, system overhead, and downstream placement impact. Section 6 discusses limitations and Section 7 concludes the paper with future work.

2. Related Work

2.1. Intent-Driven Orchestration in the Compute Continuum

Intent-Driven Orchestration (IDO) has emerged as a modern paradigm that enables decoupled application management through Service-level Objectives (SLOs), reducing administrative and operational overhead [15]. In IDO, intents express high-level policies or business goals, allowing users to specify *what* the system should achieve while the orchestration logic autonomously determines *how* to achieve it [20, 13]. Advances in Artificial Intelligence (AI) and Machine Learning (ML) further support this vision by enabling automated management of increasingly complex and dynamic compute continuum resources [21].

Despite these benefits, service placement across geographically distributed edge–cloud environments remains challenging due to fluctuating system states, resource tradeoffs, and cost constraints that must still satisfy service intents [19]. Traditional Kubernetes-based orchestration [22] lacks native intent-centric abstractions and often requires frequent manual intervention and specialized expertise, limiting its ability to adapt to dynamic continuum conditions.

Several studies extend Kubernetes and KubeEdge to automate deployment across the continuum [23, 24, 25]. However, these solutions still rely on detailed low-level configuration and significant operator involvement. At enterprise scale, managing numerous interdependent services becomes increasingly complex, and placement decisions based on transient system states remain difficult even for experienced administrators.

[1] https://doi.org/10.5281/zenodo.20810799

2.2. Intent Specification and Acquisition Approaches

Traditional intent acquisition approaches typically require users to specify lower-level SLOs directly to guide resource allocation, creating a barrier for users without expertise in system configurations.

Sebrecht et al. [1] propose a fog mesh that accepts structured intents and parses them into workflows while considering user locality and workflow constraints to ensure QoS. However, the system does not support natural-language intent expression, maintaining an entry barrier for non-expert users.

Spillner et al. [11] explore microservice control in the compute continuum using high-level business objectives. Their approach adjusts resources of already deployed services to maintain performance targets but does not determine or recommend service placement based on user intent.

Morichetta et al. [12] investigate load balancing, cost efficiency, and function coordination in serverless environments using stakeholder-defined intents. Although the work references language-model-based translation, it ultimately requires users to provide explicit SLOs rather than natural-language specifications. Their later inCoord framework [26] further shows that application-level intents in the cloud-edge continuum can be decomposed into domain-specific objectives across compute, network, and storage domains.

Filinis et al. [13] present an intent-driven orchestration framework where users supply intent descriptions through keywords and metadata such as constraints and objectives. This reliance on structured parameters and low-level configurations limits accessibility for less-experienced users.

Akbari et al. [14] introduce *iContinuum*, an intent-driven emulation toolkit that provides a realistic testbed to evaluate placement strategies under user-defined metrics such as latency, privacy, and energy consumption. The framework focuses on benchmarking placement mechanisms rather than translating natural-language intents into actionable specifications.

Metsch et al. [15] propose an orchestration architecture for cloud-native deployments in which users declare objectives through structured Kubernetes Custom Resource Definitions specifying explicit Key Performance Indicators (KPIs) and SLO targets. A planner then translates these formal specifications into scaling and tuning actions. While effective for enforcing predefined objectives, the approach assumes that users already provide precise metric-level inputs and does not derive specifications from unstructured natural-language intents. Sedlak et al. [27] diffuse formal high-level SLOs in microservice pipelines into lower-level SLOs and parameter assignments using Bayesian networks learned from runtime metrics. However, it accepts formalized goals as SLOs and therefore does not address the translation of validated placement objectives from unstructured natural-language intents.

2.3. Natural Language–Based Intent Translation

To reduce the expertise required for low-level system orchestration, recent studies have explored natural-language intent interfaces that translate human-readable intents into formal SLO specifications consumable by traditional IDO frameworks [1, 11, 12, 13, 14, 15].

Jacobs et al. [28] introduce *LUMI*, a chatbot-based interface that converts natural-language network intents into low-level configurations using a traditional named entity recognition pipeline based on word embeddings, Bi-LSTM models, and CRF tagging. While effective for short and structured utterances, the approach relies on predefined vocabularies and syntactic extraction, limiting its ability to handle complex or implicit intents and to generalize across heterogeneous environments.

Capova et al. [29] translate natural-language business intents into reinforcement learning (RL) environments through knowledge-graph retrieval and LLM-based reasoning, generating reward functions and action spaces to guide RL agents. The translation produces learning objectives rather than SLO specifications that can be directly consumed by placement frameworks, and correctness depends on training convergence without deterministic or schema-constrained guarantees.

Esashi et al. [30] propose *Action Engine*, an LLM-driven framework that converts natural-language workflow descriptions into executable FaaS workflows by selecting functions and inferring dependencies to construct a workflow DAG. The method targets application-level workflow composition and synthesis, but does not derive formal system-level specifications or placement constraints required for orchestration control.

Asif et al. [31] evaluate LLMs for natural-language intent translation in IBN policies and augment the pipeline with a KNN-based classifier to detect contradictory outputs. Since configurations are generated directly by the LLM and validated only post hoc, the process remains largely generative and lacks structured, deterministic guarantees during specification construction.

Mekrache et al. [32] translate natural-language network management intents into Network Service Descriptors (NSDs) using KB-assisted few-shot prompting, structural validation, and human feedback. Mekrache et al. [33] broaden this direction into an LLM-centric intent life-cycle architecture covering decomposition, translation, negotiation, activation, and assurance. These works target NSD generation and network intent life-cycle management, but they do not address placement-specific SLO construction where metric–operator–value constraints must be grounded against fluctuating edge–cloud resource states.

OSS-GPT [34, 35] uses assistant, planner, executor, and reporter agents to translate natural-language OSS intents into executable API-call sequences. DMO-GPT [36] extends this agentic design to distributed multi-operator 6G management by selecting operators and coordinating API execution across heterogeneous OSSs. These systems focus on direct OSS API planning and execution, where the generated

output is tied to API payloads and service descriptors. As a result, they do not provide a standalone validated SLO abstraction that can be inspected, rejected, or reused by downstream placement engines before orchestration actions are executed. Overall, these works demonstrate that natural-language interfaces can lower the expertise barrier for intent expression across domains. However, they largely rely on generative translation or post-hoc checks and do not incorporate contextual grounding or structured validation mechanisms necessary for reliable natural-language-to-SLO construction in dynamic compute-continuum environments.

2.4. Service Placement Methods

To optimize microservice placement across the compute continuum, placement algorithms are widely adopted, primarily focusing on compute and network resources [6].

Samani et al. [5] propose a coordination-based method to prevent QoS violations through Service-level Agreements (SLAs). Their approach considers both the current capabilities and historical credibility of devices when determining placement. However, Proactive Application Placement does not incorporate user intent, instead prioritizing SLA requirements defined by the infrastructure.

Mota-Cruz et al. [7] present two approaches to minimize latency and balance load, termed *app-based* and *service-based* placement. The app-based method deploys all services sequentially within an application, whereas the service-based method evaluates each service independently while accounting for interconnected loads. Although the study compares different strategies based on application context and optimization objectives, it does not consider high-level user intent to guide placement decisions in the compute continuum.

2.5. Retrieval-Augmented Grounding for LLM-Based Systems

LLM-based intent translation can be improved either through model adaptation or inference-time grounding. While large-scale pretraining and domain-specific fine-tuning improve specialization, they are costly to maintain in rapidly changing system environments.

Inference-time methods condition the model without modifying its parameters. Zero-shot prompting [37] and few-shot in-context learning [38] provide lightweight adaptation, while retrieval-augmented generation (RAG) grounds responses using external context [39]. This is particularly relevant for distributed-system intent translation, where implicit requirements often depend on fluctuating runtime metrics; retrieving live or historical continuum state enables concrete value resolution instead of prompt-only estimation.

Several recent approaches use retrieval and reasoning for executable task generation. Zhang et al. [40] propose *Reverse Chain*, which decomposes requests into API selection and argument completion. Yao et al. [41] introduce *ReAct*, which interleaves reasoning with environment actions to retrieve evidence during decision making. Verma et al. [42] propose *Plan-RAG*, which decomposes queries into a DAG of sub-tasks for targeted retrieval and generation.

These approaches improve grounded reasoning and reduce hallucination, but they primarily target workflow planning, API generation, or question answering. They do not directly construct formally constrained system-level specifications for orchestration, where retrieved evidence must be combined with schema validation and control-plane compatibility.

2.6. Research Gaps

Existing IDO frameworks typically require users to provide explicit SLO specifications, shifting complexity to users who need domain expertise and contextual system knowledge. Natural-language intent interfaces reduce this barrier, but existing approaches remain limited for compute-continuum placement because they often depend on direct prompt-based generation or post-hoc validation. Such approaches provide limited support for intents whose correct specification depends on monitored infrastructure state, such as "highest bandwidth" or "lowest memory utilization".

A gap therefore remains in reliably transforming natural-language placement intents into orchestration-consumable SLO artifacts before they are used by downstream placement logic. This requires not only language interpretation, but also schema-bounded specification construction, infrastructure-aware grounding, and rejection of unsupported, ambiguous, or conflicting requests. This paper addresses this gap through *Intent Engine*.

3. System Architecture

This section presents the proposed *Intent Engine* architecture as a control-plane SLO construction system for the compute continuum. It is designed to integrate with existing intent-driven orchestration (IDO) and placement frameworks without replacing their placement algorithms or runtime assurance mechanisms.

Reliable SLO construction is critical because the translated output is intended to become an orchestration-consumable control-plane artifact for a downstream IDO framework. If an intent is mistranslated, the downstream framework may receive unsupported constraints, incorrect resource values, conflicting objectives, or malformed specifications. Such errors can lead to unintended service placement, SLO violations, inefficient resource use, or application disruption once the specification is consumed by an orchestrator. Therefore, *Intent Engine* treats intent translation as a controlled specification-construction process rather than unconstrained end-to-end text generation.

Figure 1 illustrates the proposed architecture. The pipeline first extracts semantic constraints from the natural-language intent into an intermediate SLO representation (IR). It then grounds implicit or context-dependent requirements by retrieving evidence from the compute-continuum infrastructure state. Finally, structural and schema validation is applied before producing the final SLO specification, ensuring compatibility with existing orchestration and placement engines. *Intent Engine* is limited to intent acquisition and SLO construction; placement decisions, deployment

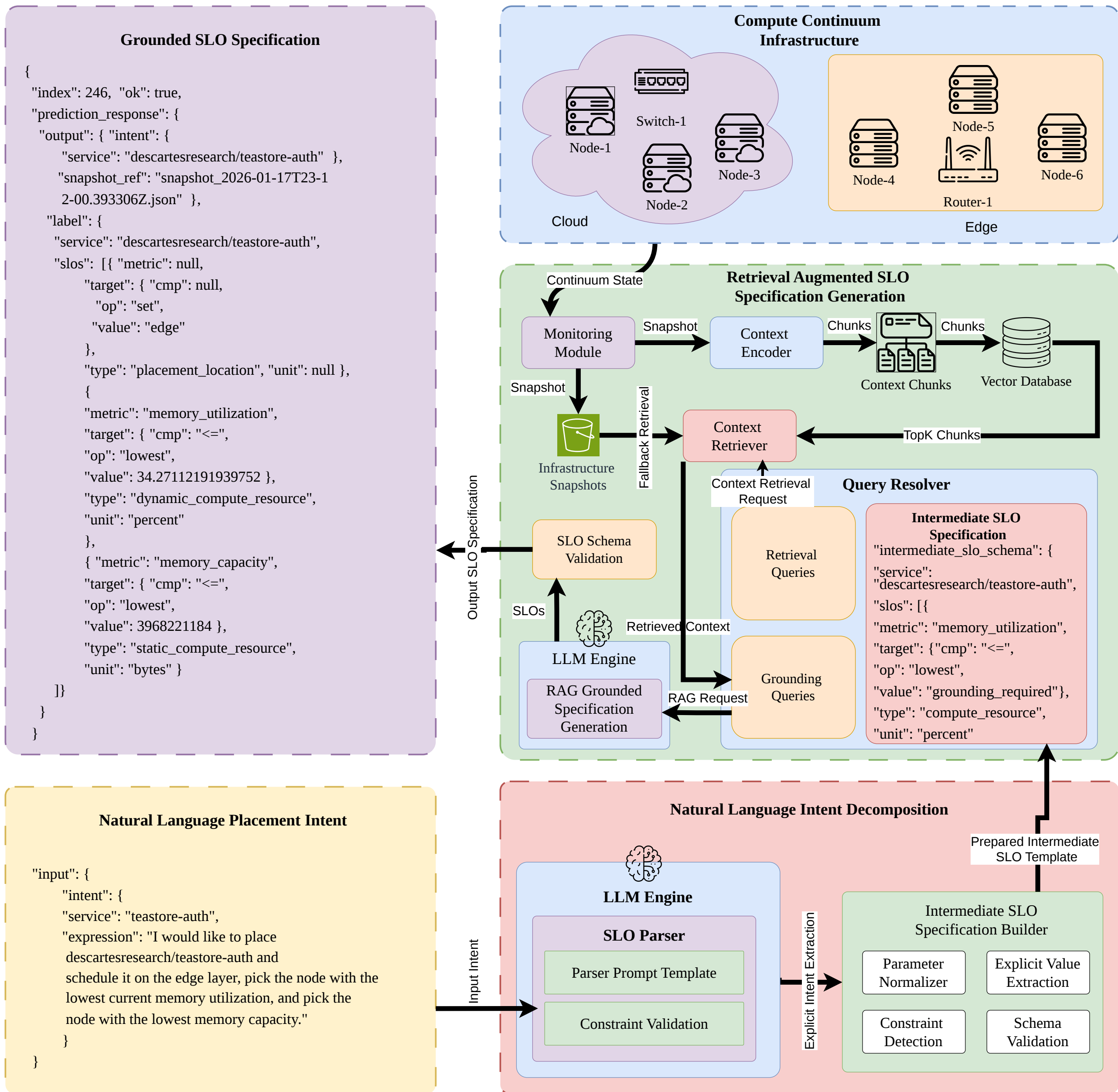

Figure 1: Overview of the proposed *Intent Engine* system architecture.

execution, runtime intent assurance, and re-grounding are handled by the downstream IDO framework that consumes the generated specification.

The architecture consists of two primary components: (i) *Natural Language Intent Decomposition* and (ii) *Retrieval-Augmented SLO Specification Generation*. The former extracts explicit constraints, identifies implicit placement requirements, and prepares an intermediate SLO representation. The latter grounds implicit requirements through contextual system-state retrieval and produces the final validated SLO specification. Together, these components form the artifact boundary between user intent and downstream orchestration logic: only schema-valid and infrastructure-grounded SLO specifications are emitted. The core modules are described below.

3.1. System Input

The architecture accepts two inputs for each placement request: the *service name* and a natural-language *intent expression*. The intent may describe explicit constraints (e.g., "memory utilization below 60%"), implicit preferences (e.g., "highest memory utilization"), or deployment location requirements (e.g., "deploy in the cloud").

To ensure compatibility with existing IDO frameworks [1, 11, 12, 13, 14, 15], we define a supported SLO schema

shown in Table 2. This infrastructure-oriented schema is consistent with inCoord by Morichetta et al., where application-level intents are decomposed into domain-specific compute, network, and storage objectives. The selected constraint types reflect the acquisition specifications commonly used by existing IDO and placement frameworks, including placement location, compute capacity, utilization, and network-related constraints. The schema is also configurable, allowing framework-specific metrics, units, and constraint types to be added when required. Schema validation checks whether each generated SLO uses only supported metrics, operators, units, and value formats before the specification is emitted. Restricting translation to this predefined schema preserves downstream compatibility, limits the LLM output space, and reduces unsupported or hallucinated constraints. This design enables *Intent Engine* to operate as a flexible front-end intent acquisition component for existing frameworks.

3.2. Natural Language Intent Decomposition

The intent decomposition component transforms an unstructured natural-language intent into a structured, schema-compliant intermediate SLO representation. It consists of two modules: the *SLO Parser* and the *Intermediate SLO Specification Builder*. Unlike prior approaches that directly generate final configuration or policy artifacts [29, 31], our architecture separates intent understanding, constraint extraction, and schema construction before contextual grounding.

3.2.1. SLO Parser

The SLO Parser extracts constraints from the service-specific natural-language intent using an LLM guided by the parser prompt template shown in Figure 7. The prompt directs the model to identify explicitly stated metrics, comparison operators, and values while adhering to the supported SLO schema and normalized unit formats.

This stage focuses on information explicitly present in the intent and prepares placeholders for requirements that need contextual grounding. The prompt enforces: (i) adherence to predefined SLO categories (Table 2), (ii) canonical unit and operator normalization, (iii) a fixed intermediate SLO schema, and (iv) exclusion of unsupported parameters.

Because the IR is extracted by an LLM, it may still include unsupported capabilities, malformed units or operators, or conflicting constraints. To prevent these errors from propagating, the SLO Parser applies *Constraint Validation* using Algorithm 1. The algorithm checks the generated IR against the supported capability registry and rejects invalid or inconsistent clauses before grounded SLO construction, preventing unsafe intermediate outputs from reaching downstream orchestration logic.

3.2.2. Intermediate SLO Specification Builder

The *Intermediate SLO Specification Builder* post-processes the parser output into a schema-compliant intermediate representation for contextual grounding. It normalizes metrics, operators, and units; maps explicit values to canonical fields; verifies conformity with the supported SLO schema; and marks constraints without explicit values as requiring grounding.

This intermediate representation acts as a control point between language understanding and final specification generation. By decomposing the intent into validated components, the architecture avoids monolithic LLM generation and constructs the final specification through guided, context-aware reasoning.

3.3. Retrieval-Augmented SLO Specification Generation

This component takes the intermediate SLO representation and grounds implicit requirements by retrieving contextual information from the compute continuum. It addresses cases where prompt-only generation is insufficient, particularly extrema-based requests such as “highest memory” or constraints that depend on fluctuating system conditions.

3.3.1. Query Resolver

The Query Resolver identifies constraints that require contextual grounding and formulates retrieval queries specifying the metric, scope, and grounding objective, such as highest or lowest resource value. It then combines the intermediate SLO template with retrieved evidence to guide grounded value generation while preserving the predefined schema.

3.3.2. Context Retriever and Encoder

The Context Retriever gathers compute-continuum information from two sources: (i) a vector database containing embedded system-state representations and (ii) a fallback snapshot stored in file storage. Retrieval first performs a top-k similarity search on the vector database[2]; if the retrieved evidence is insufficient or unavailable, the system falls back to snapshot-based context retrieval.

The Context Encoder converts infrastructure snapshots into retrieval-ready chunks containing node-level states, aggregated edge–cloud summaries, and structured resource metadata. The encoding process is deterministic and does not rely on an LLM, ensuring consistent context representation and avoiding additional generative errors in the grounding source.

3.3.3. Monitoring Module

The monitoring module continuously collects static and dynamic resource metrics across edge and cloud nodes, including compute capacities, utilization levels, and network statistics. Metrics are sampled periodically[3] and stored as both raw snapshots and encoded representations for retrieval. This module provides the infrastructure state required for reliable grounding.

3.4. Compute Continuum Infrastructure

The compute continuum infrastructure comprises heterogeneous edge and cloud resources managed by existing orchestration frameworks. In this work, the infrastructure

[2] $k = 8$ in our implementation.
[3] every 10 minutes in our implementation.

Table 1
Compute resource specification of the compute continuum testbed.

Node	Model	CPU	Memory	Storage	Role	Location
Node 1	Raspberry Pi 4	2 cores	4 GB	64 GB	Worker	Edge
Node 2	AWS EC2 Instance	2 cores	2 GB	128 GB	Worker	Cloud
Node 3	Raspberry Pi 5	4 cores	8 GB	64 GB	Master	Edge
Node 4	Raspberry Pi 4	4 cores	4 GB	64 GB	Worker	Edge
Node 5	Raspberry Pi 3	2 cores	4 GB	32 GB	Worker	Edge
Node 6	AWS EC2 Instance	2 cores	2 GB	80 GB	Worker	Cloud

primarily supplies the contextual state required for grounding placement intents and constructing ground-truth SLO labels. The proposed architecture is agnostic to the underlying infrastructure implementation, enabling integration with a wide range of edge–cloud platforms and orchestration systems. Evaluating the placement decisions made after an IDO framework consumes the generated SLOs is outside the scope of the translation layer studied here.

4. Implementation and Experimental Setup

We implemented *Intent Engine* as an intent-to-SLO construction layer and evaluated its accuracy, robustness, and grounding effectiveness. Experiments were conducted on a real compute-continuum testbed, enabling systematic comparison with LLM-based intent translation baselines under realistic edge–cloud conditions.

The implementation consists of two main components: (i) a compute continuum testbed spanning edge and cloud layers, and (ii) the *Intent Engine* pipeline for intent decomposition, retrieval-augmented grounding, and SLO specification generation. The testbed supplies periodically monitored system context for grounding implicit constraints and deriving labeled SLO targets, while the pipeline executes the SLO construction process described in Section 3. The testbed is not used to evaluate downstream placement quality or runtime QoS optimization, which depend on the IDO framework that consumes the generated SLOs. Each component is described below.

4.1. Compute Continuum Infrastructure Testbed

We built a physical compute continuum testbed spanning geographically distributed edge and cloud resources to collect realistic resource-state traces under deployment-like conditions and resource variability. Since retrieval-augmented grounding depends on current system context, the infrastructure continuously collects resource-state data used by the SLO grounding pipeline and by the construction of ground-truth labels.

The testbed comprises heterogeneous compute and network resources interconnected across the edge–cloud continuum, as shown in Figure 2. Table 1 summarizes the hardware specifications of the deployed nodes.

All six nodes are managed as a single Kubernetes [22] cluster spanning edge and cloud locations. Node 3 operates as the master node responsible for cluster orchestration and scheduling, while the remaining nodes act as workers interconnected through Router 1 and Router 2 to expose real network variability.

The edge layer consists of heterogeneous Raspberry Pi devices, providing diverse resource capacities for trace collection and grounding. The cloud layer is provisioned using Amazon Web Services (AWS) in the *us-east-1* region, where EC2 [43] instances are configured within a dedicated *EC2 Security Group* and integrated as Kubernetes worker nodes. Although AWS is used in our deployment, the architecture remains cloud-provider agnostic.

An External Management Host is used for Kubernetes cluster management, infrastructure control, and visualization of continuum-state analytics, supporting the implementation of the *Monitoring Module*. This host is external to the compute continuum; service placement and orchestration are executed within the cluster by the master node. *Intent Engine* runs on the External Management Host, and its internal architecture includes the LLM engine, intent decomposition, context retrieval, and retrieval-grounded SLO generation components.

Network connectivity spans distinct access routers to emulate distributed domains. A secure virtual private network (VPN) overlay is established using Tailscale [44] to enable private communication between edge and cloud nodes. To emulate realistic deployment scenarios, we deploy the TeaStore [45] reference microservice application, comprising six interacting services, using Kubernetes-native deployment.

4.2. Natural Language Intent Decomposition

In our implementation, service placement intents are supplied as unstructured natural-language inputs and processed by the intent decomposition component to produce structured intermediate specifications.

4.2.1. SLO Parser

The *SLO Parser* is implemented using an LLM engine guided by the parser prompt template shown in Figure 7. We use GPT-4.1-mini through the OpenAI [46] API as the underlying model. The temperature is set to 0.0 to preserve extraction stability and reduce unnecessary output variation.

The parser is implemented in Python and exposed as a lightweight Flask [47]-based API service for integration with the orchestration pipeline. The API accepts JavaScript Object Notation (JSON) requests containing the service identifier and intent text, and returns a structured JSON

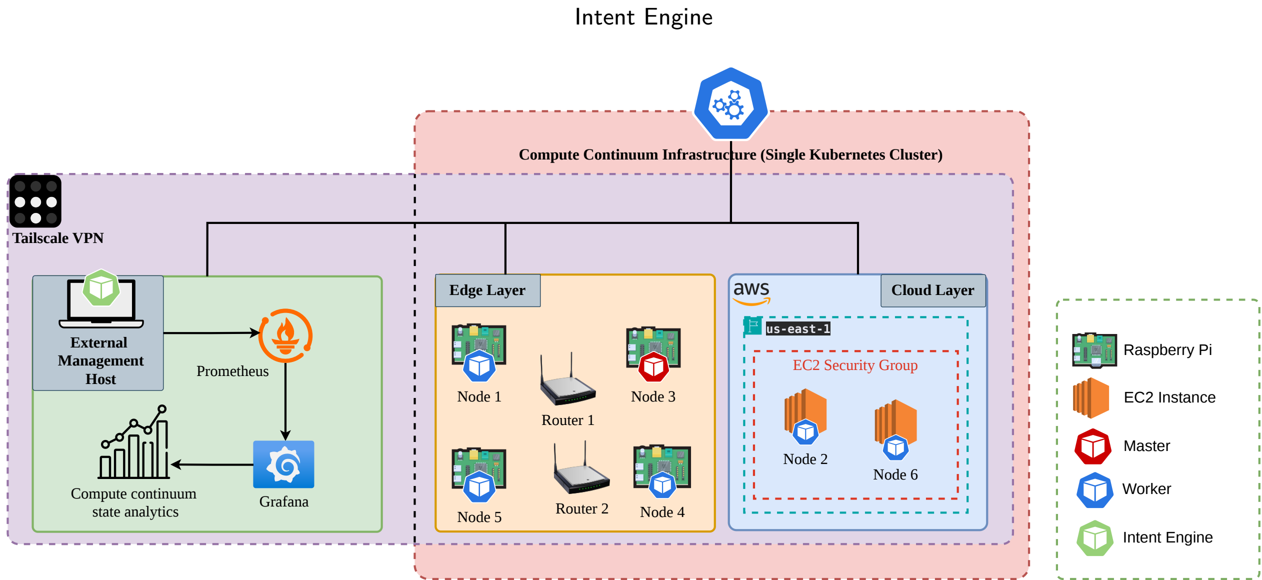

Figure 2: Overview of the real compute continuum testbed and resource monitoring infrastructure.

Table 2
Supported constraint space used by the generated IR and downstream grounding. Each capability c is associated with an admissible domain $\mathcal{D}(c)$ and a canonical representation $\bar{d}(c)$.

Category	Capability c	Admissible Domain $\mathcal{D}(c)$	Canonical Representation $\bar{d}(c)$
Placement	Placement Location	Edge, Cloud, Both, Any	Edge / Cloud / Both / Any
Static Compute	CPU Capacity	Cores, Millicores	Cores
	Memory Capacity	Bytes, B, KB, MB, GB, KiB, MiB, GiB	Bytes
	Storage Capacity	Bytes, B, KB, MB, GB, KiB, MiB, GiB	Bytes
Dynamic Compute	CPU Utilization	%	Percent
	Memory Utilization	%	Percent
	Storage Utilization	%	Percent
Node Network	Bandwidth	Bps, Kbps, Mbps, Gbps	Bps
	Port Utilization	%	Percent

response that is forwarded to subsequent stages for normalization and grounding.

4.2.2. *Intermediate SLO Specification Builder*

This module post-processes the parser output to construct a schema-compliant intermediate SLO representation. It enforces the predefined schema by normalizing field names, inserting missing fields with null values, and validating placement and parameter constraints. It also canonicalizes metric names, units, and operators, and annotates each constraint with a `grounding_required` flag to indicate whether contextual grounding is needed.

The resulting intermediate specification is serialized as a JSON artifact and forwarded to the retrieval-augmented grounding stage.

4.3. Retrieval-Augmented SLO Specification Generation

The retrieval-augmented generation (RAG) component grounds intent constraints using monitored compute continuum context. It accepts the intermediate specification as a JSON artifact, detects parameters requiring grounding, retrieves contextual information, generates final SLO specifications, and validates structural and schema completeness.

4.3.1. *Monitoring Module*

We monitor the compute continuum testbed using a kube-prometheus stack. Prometheus [48] collects static resource capacities and dynamic utilization metrics, including CPU, memory, storage, and network statistics.

Grafana [49] is used for visualization and manual inspection of resource trends. Additionally, we implement a Python Flask [47]-based API service that periodically queries Prometheus through node-exporter every 10 minutes, extracts point-in-time infrastructure snapshots, and stores them as JSON artifacts in Amazon S3 [50]. These snapshots serve as a deterministic fallback source for contextual grounding.

Each snapshot is also forwarded to the *Context Encoder* for embedding and storage in the vector database.

Algorithm 1 LLM-Generated IR Validation

Input: LLM-generated IR I, supported capability registry $\mathcal{R}$
Output: validated intent IR I_{valid}
1: $I_{\text{valid}} \leftarrow [\,]$, $p \leftarrow$ `Any`
2: initialize $lb[c] \leftarrow -\infty$, $ub[c] \leftarrow +\infty$, $ext[c] \leftarrow \varnothing$ for each numeric capability $c \in \mathcal{R}$ ▷ $lb[c], ub[c]$: feasible bounds; $ext[c]$: extremum request
3: **for all** generated clause $s \in I$ **do**
4: $(c, r, v) \leftarrow$ NORMALIZE$(s, \mathcal{R})$ ▷ c: capability; r: relation; $v \in \bar{d}(c)$
5: **if** (c, r, v) is invalid **then**
6: **return** $\emptyset$ ▷ unsupported capability or invalid unit/value
7: **end if**
Non-numeric validation
8: **if** $c =$ `Placement Location` **then**
9: **if** $v =$ `Any` **then**
10: append s to I_{valid}; **continue** ▷ no placement restriction
11: **else if** $p =$ `Any` or $p = v$ **then**
12: $p \leftarrow v$
13: **else if** $p =$ `Both` or $v =$ `Both` **then**
14: $p \leftarrow$ `Both`
15: **else if** ($p =$ `Edge` $\wedge$ $v =$ `Cloud`) or ($p =$ `Cloud` $\wedge$ $v =$ `Edge`) **then**
16: $p \leftarrow$ `Both` ▷ merge Edge and Cloud
17: **else**
18: **return** $\emptyset$ ▷ conflicting placement values
19: **end if**
20: append s to I_{valid}; **continue**
21: **end if**
Numeric validation
22: **if** $r =$ `>=` or $r =$ `=` **then**
23: $lb[c] \leftarrow \max(lb[c], v)$
24: **end if**
25: **if** $r =$ `<=` or $r =$ `=` **then**
26: $ub[c] \leftarrow \min(ub[c], v)$
27: **end if**
28: **if** $lb[c] > ub[c]$ **then**
29: **return** $\emptyset$ ▷ empty feasible interval
30: **end if**
31: **if** $r =$ `max` **then**
32: **if** $ext[c] =$ `min` **then**
33: **return** $\emptyset$ ▷ opposite extrema
34: **end if**
35: $ext[c] \leftarrow$ `max`
36: **else if** $r =$ `min` **then**
37: **if** $ext[c] =$ `max` **then**
38: **return** $\emptyset$ ▷ opposite extrema
39: **end if**
40: $ext[c] \leftarrow$ `min`
41: **end if**
42: append s to I_{valid}
43: **end for**
44: **return** I_{valid} ▷ forward only supported and consistent IR

4.3.2. *Context Encoder*

The Context Encoder processes each infrastructure snapshot into retrieval-ready context chunks. Each chunk captures node-level state, network measurements, or aggregated summaries, and is encoded as embedding-friendly text paired with structured metadata. The chunks are stored in a Qdrant [51] vector database to support similarity-based retrieval.

Chunking is implemented deterministically rather than through LLM-based generation to ensure reproducibility and avoid introducing spurious context. During retrieval, only the top-k most relevant chunks are returned; in our implementation, $k = 8$.

4.3.3. *Context Retriever*

The Context Retriever resolves metric queries using a two-stage strategy. Queries are first executed against the Qdrant vector database using similarity search to obtain relevant chunks. If retrieval is insufficient or unavailable, the system falls back to the most recent snapshot stored in Amazon S3 [50].

Retrieved values are then aggregated deterministically, with unit normalization and metadata filtering applied as required.

4.3.4. *Query Resolver*

The Query Resolver analyzes the intermediate SLO specification and identifies parameters requiring contextual grounding. For each parameter, it generates structured retrieval queries that encode the desired aggregation semantics, such as highest or lowest.

Retrieved context is combined with grounding prompts and passed to the LLM engine to instantiate concrete values. Explicitly specified constraints bypass retrieval and are propagated directly. The grounded outputs are then canonicalized to a fixed schema, with metric names, units, and operators normalized to match the conventions used in the infrastructure snapshots, ensuring consistent grounding and stable specification construction.

4.3.5. *SLO Specification Validation*

The final SLO specification is validated for structural completeness, schema consistency, and value correctness. Units and field names are checked against the snapshot representations to guarantee alignment with the monitored system state. The validated specification is serialized as a JSON artifact and forms the final system output.

5. Evaluation

This section evaluates whether *Intent Engine* can reliably construct accurate SLO artifacts from natural-language placement intents for downstream orchestration. The evaluation focuses on the intent acquisition and SLO construction layer rather than runtime orchestration behavior. Specifically, it examines: (i) whether generated SLO constraints match the ground truth, (ii) whether the complete specification is structurally correct, (iii) whether hallucinations are reduced, (iv) whether retrieval grounding improves implicit value resolution, (v) whether invalid intents are safely rejected, (vi) whether the construction pipeline operates within acceptable latency and scalability limits, and (vii) how translation errors affect downstream placement feasibility.

5.1. Dataset

To the best of our knowledge, no public dataset exists for grounded natural-language intent-to-SLO translation in a compute-continuum environment. We therefore construct a dataset from our real testbed and augment it with synthetic natural-language variants to increase linguistic diversity while preserving the target SLO schema

Table 3
Dataset composition by intent complexity level, source type, validity status, and number of SLOs per valid intent.

Complexity Level	Total	Real	Synthetic	Valid	Invalid	SLOs per Valid Intent
Level 1	189	59	130	137	52	1
Level 2	207	67	140	174	33	2–3
Level 3	177	57	120	141	36	2–3
Level 4	93	23	70	48	45	3–4
Level 5	50	10	40	21	29	2–3
Total	**716**	**216**	**500**	**521**	**195**	–

Table 3 depicts the dataset containing 716 intent records across five complexity levels. Of these, 521 records are valid intent-to-SLO pairs, and 195 records are invalid cases covering ambiguous, conflicting, malformed, and unsupported intents. Valid records are used for translation-correctness evaluation, while invalid records support robustness and failure-handling analysis. Each record is associated with a point-in-time compute-continuum snapshot used to derive the ground-truth labels and support contextual grounding during evaluation.

Real records are derived from the physical edge–cloud testbed using monitored infrastructure snapshots. Synthetic records are generated using the Llama 3.3 70B model to expand the linguistic diversity of intents associated with the real testbed traces. This augmentation is used only to increase natural-language variation while preserving the same schema, supported metrics, and complexity structure. To avoid synthetic-label noise, all records are checked against the supported SLO schema, and implicit values are derived from the corresponding snapshots rather than accepted from the generation model. Invalid records are retained rather than discarded so that the evaluation also captures ambiguous, conflicting, malformed, and unsupported user intents.

The TeaStore microservice application is used only as a reference workload to generate realistic service identifiers, placement scenarios, and infrastructure traces. The proposed architecture is application-agnostic: *Intent Engine* constructs SLO artifacts over the supported schema and monitored infrastructure state, regardless of the specific application used to produce the traces. TeaStore therefore provides a concrete service context for dataset construction, but the evaluated task is not TeaStore-specific.

Complexity levels reflect the number and type of constraints in an intent. Level 1 contains a single SLO. Levels 2–4 contain increasing combinations of placement, compute, and network constraints. Level 5 contains the most challenging cases, where implicit snapshot-grounded requirements are often combined with explicit placement or threshold constraints. Dataset examples, including valid and invalid records across complexity levels, are illustrated in Figure 10 in the Appendix.

5.2. Candidate Models

Table 4 summarizes the LLMs used in dataset construction and evaluation. Llama 3.3 70B is used only for synthetic intent generation, while GPT-4.1 mini, Claude Sonnet 4.5, and DeepSeek V4-Flash are used as translation evaluation backends. This separation reduces model-bias risk because the model used to diversify the synthetic intents is not used to evaluate the architecture. Using multiple closed- and open-source evaluation backends further tests whether the observed gains generalize across model families rather than depending on a single LLM.

Table 4
Candidate LLMs used for synthetic dataset generation and translation evaluation.

Model	Access	Role
Llama 3.3 70B	Open	Synthetic intent generation.
GPT-4.1 mini	Closed	Translation evaluation backend.
Claude Sonnet 4.5	Closed	Translation evaluation backend.
DeepSeek V4-Flash	Open	Translation evaluation backend.

5.3. Baseline Methods

To the best of our knowledge, no prior system directly translates unstructured natural-language placement intents into retrieval-grounded orchestration-consumable SLO specifications for compute-continuum environments. We therefore compare *Intent Engine* against four prompt-only LLM baselines commonly used in recent natural-language intent translation work [29, 30]: *Zero-shot*, *Few-shot*, *Zero-shot Chain-of-Thought (CoT)*, and *Few-shot CoT*. Task-specific pretrained or fine-tuned baselines are not included because no established labeled dataset or pretrained model exists for this domain-specific grounded intent-to-SLO translation task.

To include a non-prompting structured alternative, we implement a *Rule-based Parser* baseline. It uses schema constrained metric, operator, placement, and unit rules with snapshot resolution for implicit highest/lowest constraints. The parser uses no prompts, LLM outputs, learned parameters, or dataset-specific templates, making it a generic non-LLM baseline under the same SLO schema.

For evaluation fairness, the prompting baselines are also provided with the infrastructure snapshots when grounded values are required. This gives the baselines access to the same contextual evidence, although it increases prompt length and complexity compared with the retrieval and schema-bounded pipeline used by *Intent Engine*. This baseline setting differs from *Intent Engine*, which retrieves only metric- and intent-relevant infrastructure context instead of

injecting the full continuum snapshot into a single prompt. This reduces unnecessary context length and avoids exposing the LLM to unrelated node-state information.

We evaluate the baselines and *Intent Engine* using three backend LLMs: GPT-4.1 mini, Claude Sonnet 4.5, and DeepSeek V4-Flash. The same prompt design is applied consistently across backends. The prompt templates for the baseline methods are shown in Figure 8 and Figure 9. The ablation study uses GPT-4.1 mini to isolate the effect of retrieval grounding while holding the backend model fixed.

5.4. Metrics and Definitions

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ denote the dataset, where $x_i = (s_i, t_i)$ consists of a service identifier s_i and a natural-language intent expression t_i, and y_i is the corresponding ground-truth SLO specification. The dataset contains $N = 716$ records, with a valid subset $\mathcal{D}_{\text{valid}}$ of 521 valid intent-to-SLO pairs. Unless otherwise stated, translation-correctness metrics are computed on $\mathcal{D}_{\text{valid}}$.

Each record i is associated with a system-state snapshot c_i captured from the testbed at labeling time. This snapshot is not provided as a user input; it is used to derive ground-truth labels and support contextual grounding during evaluation.

For each record i, we represent the ground-truth SLO specification as a set of atomic constraints $\mathcal{S}_i^{\text{gt}}$ and the predicted specification as $\mathcal{S}_i^{\text{pred}}$. Each atomic constraint is canonicalized as a tuple of resource type, metric, operator/comparator, value, and normalized unit. A predicted constraint is counted as a true positive (TP) when it exactly matches a ground-truth constraint under this canonical representation. A false positive (FP) is a predicted constraint with no matching ground-truth constraint, and a false negative (FN) is a ground-truth constraint missing from the prediction.

We use strict matching rather than mean absolute error (MAE) or tolerance-based scoring because SLOs and SLAs are threshold-based control-plane artifacts that can be consumed by downstream placement or orchestration frameworks. A small numeric deviation can still change whether a constraint is satisfied or violated, and therefore can affect downstream placement decisions. Exact matching better reflects the reliability requirement of validated SLO construction.

For a method m and complexity level l, we aggregate true positives ($TP_{m,l}$), false positives ($FP_{m,l}$), and false negatives ($FN_{m,l}$), and compute:

$$\text{Precision}_{m,l} = \frac{TP_{m,l}}{TP_{m,l} + FP_{m,l}}, \tag{1}$$

$$\text{Recall}_{m,l} = \frac{TP_{m,l}}{TP_{m,l} + FN_{m,l}}, \tag{2}$$

$$\text{F1}_{m,l} = \frac{2 \cdot \text{Precision}_{m,l} \cdot \text{Recall}_{m,l}}{\text{Precision}_{m,l} + \text{Recall}_{m,l}}. \tag{3}$$

For specification-level structure, we use Exact Match and Jaccard similarity:

$$\text{ExactMatch}(i) = \begin{cases} 1, & \text{if } \mathcal{S}_i^{\text{pred}} = \mathcal{S}_i^{\text{gt}}, \\ 0, & \text{otherwise,} \end{cases} \tag{4}$$

$$\text{Jaccard}(i) = \frac{\left|\mathcal{S}_i^{\text{pred}} \cap \mathcal{S}_i^{\text{gt}}\right|}{\left|\mathcal{S}_i^{\text{pred}} \cup \mathcal{S}_i^{\text{gt}}\right|}. \tag{5}$$

To quantify hallucinations, we track three binary indicators per response:

$$h_i^{(1)} = \mathbb{1}\{\text{unsupported constraint occurs in response } i\},$$
$$h_i^{(2)} = \mathbb{1}\{\text{incorrect or missing value in response } i\},$$
$$h_i^{(3)} = \mathbb{1}\{\text{schema violation occurs in response } i\}.$$

Average rates are:

$$H_k = \frac{1}{N} \sum_{i=1}^{N} h_i^{(k)}, \quad k \in \{1, 2, 3\}, \tag{6}$$

and the aggregate hallucination score is

$$H = H_1 + H_2 + H_3, \tag{7}$$

where H is the average number of hallucination categories triggered per response.

For invalid intents, the desired behavior is to reject the request or return no orchestration-consumable SLO specification. We therefore define the rejection rate as:

$$\text{RejectionRate}_{m,c} = \frac{N_{m,c}^{\text{reject}}}{N_c^{\text{invalid}}}, \tag{8}$$

where $N_{m,c}^{\text{reject}}$ denotes the number of invalid intents in category c correctly rejected by method m, and N_c^{invalid} denotes the total number of invalid intents in that category. Higher values indicate safer failure handling. A false accept occurs when an invalid intent is incorrectly converted into one or more validated SLO constraints.

For downstream placement-impact analysis, we use the deterministic placement generator as an evaluator. Let N_{feasible} denote the number of valid records whose ground-truth SLO label has at least one feasible placement in the corresponding snapshot. For a method m, let $N_{\text{no_match}}^m$ denote the number of cases where the predicted SLOs produce no matching node, and let N_{invalid}^m denote the number of cases where the predicted SLOs return a node that does not satisfy at least one constraint in the ground-truth SLO label. The placement failure rate is defined as:

$$\text{PlacementFailure}_m = \frac{N_{\text{no_match}}^m + N_{\text{invalid}}^m}{N_{\text{feasible}}}. \tag{9}$$

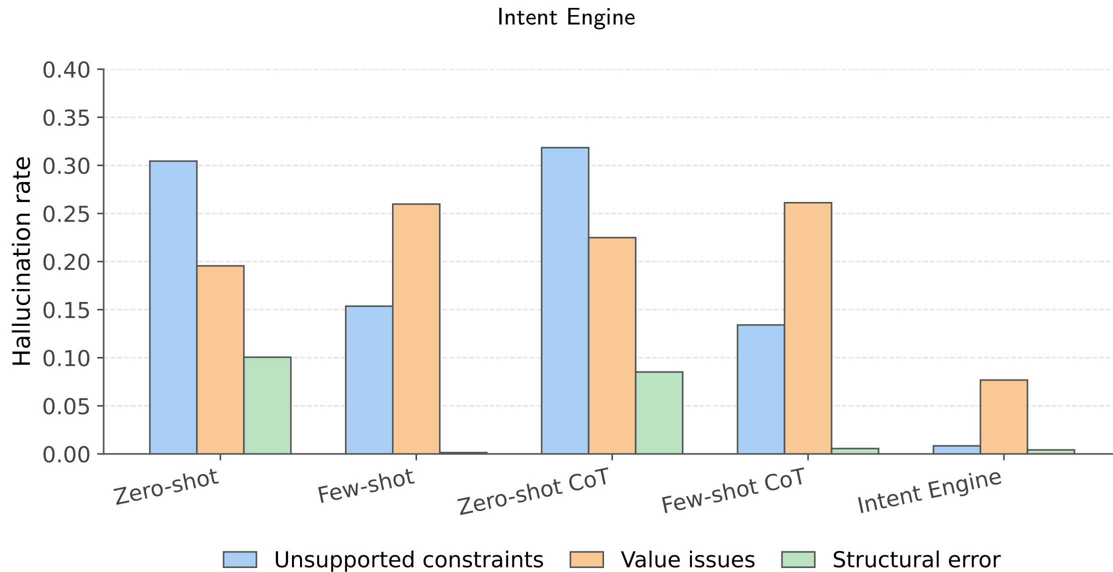

Figure 3: Overall hallucination rates across baselines using GPT-4.1 mini.

Table 5
Overall hallucination rate by type across baseline translation methods using GPT-4.1 mini.

Method	H_1	H_2	H_3	H
Zero-shot	0.3045	0.1955	0.1006	0.6006
Few-shot	0.1536	0.2598	0.0014	0.4148
Zero-shot CoT	0.3184	0.2249	0.0852	0.6285
Few-shot CoT	0.1341	0.2612	0.0056	0.4008
Intent Engine	**0.0084**	**0.0768**	0.0042	**0.0894**

5.5. Hallucination Analysis

We compute hallucination rates over the full dataset of $N = 716$ records, including both valid and invalid intents. Correct rejection of an invalid intent does not trigger a hallucination category; if an invalid intent is accepted and converted into SLO constraints, the output is evaluated using the same canonical representation. H_1 captures unsupported or spurious constraints, H_2 captures missing or incorrect values for otherwise matched constraints, and H_3 captures structural or schema violations such as invalid resource types, malformed operators, or inconsistent units.

Figure 3 and Table 5 show that the Intent Engine yields the lowest aggregate hallucination rate across translation methods. Zero-shot prompting produces the highest aggregate score, mainly due to unsupported constraints and incorrect or missing values. Few-shot and CoT prompting reduce some unsupported-constraint and schema errors, but still exhibit value errors.

Intent Engine reduces aggregate hallucination H by 85.1% relative to Zero-shot (0.6006→0.0894). It also reduces unsupported or spurious constraints by 97.2% (H_1: 0.3045→0.0084), value issues by 60.7% (H_2: 0.1955→0.0768), and structural errors by 95.8% (H_3: 0.1006→0.0042). The remaining hallucinations are mostly value-related, showing that grounded value resolution is the hardest part of the SLO construction process.

5.6. Retrieval-Grounding Ablation

To isolate the effect of retrieval grounding, we remove the RAG module and evaluate the pipeline without snapshot-derived value resolution. In this ablated setting, the system can still extract explicit constraints such as placement requirements and numeric thresholds, but it cannot resolve implicit requirements such as highest or lowest resource/utilization values from the infrastructure state.

Table 6 shows that removing retrieval grounding mainly affects higher-complexity intents. When all SLOs are considered, the no-RAG variant still receives partial credit because many high-complexity records combine implicit requirements with explicit placement or numeric-threshold constraints. However, when the evaluation is restricted to implicit snapshot-dependent constraints only, the no-RAG variant collapses to zero F1 across Levels 2–5. With RAG enabled, the Intent Engine recovers nonzero F1 on these implicit constraints, demonstrating that retrieval is essential for resolving grounded highest/lowest values.

5.7. Correctness of Generated SLO Specifications

We evaluate correctness at two levels. Contextual accuracy measures whether each generated SLO constraint matches the ground truth after canonicalization of type, metric, operator, value, and unit. Structural accuracy measures whether the complete predicted SLO set matches or overlaps with the ground-truth specification.

5.7.1. Contextual Accuracy

We compute TP, FP, and FN through strict canonical constraint matching and derive Precision, Recall, and F1 using Eqs. 1–3. Table 7 reports the overall F1 Scores across model families and data sources.

Table 6
Retrieval-grounding ablation results for Intent Engine with and without RAG using GPT-4.1 mini model.

Variant	Scope	L1	L2	L3	L4	L5
No RAG	Explicit + implicit	1.000	0.926	0.775	0.604	0.500
RAG	Explicit + implicit	1.000	0.979	0.940	0.863	0.857
No RAG	Implicit only	–	0.000	0.000	0.000	0.000
RAG	Implicit only	–	0.691	0.705	0.619	0.667

Table 7
Overall F1 Score comparison across model families and data sources for baseline methods.

Model	Source	Zero-shot	Few-shot	Zero-shot CoT	Few-shot CoT	Intent Engine
GPT-4.1 mini	Real	0.296	0.711	0.555	0.716	**0.993**
	Synthetic	0.345	0.768	0.598	0.775	**0.920**
	Total	0.331	0.751	0.585	0.758	**0.941**
Claude Sonnet 4.5	Real	0.296	0.729	0.729	0.737	**0.878**
	Synthetic	0.348	0.777	0.822	0.795	**0.867**
	Total	0.333	0.763	0.794	0.778	**0.870**
DeepSeek V4-Flash	Real	0.294	0.715	0.731	0.711	**0.869**
	Synthetic	0.344	0.776	0.799	0.774	**0.916**
	Total	0.329	0.758	0.779	0.756	**0.903**

Table 7 shows that the Intent Engine achieves the highest F1 Score for every model family and data source. For GPT-4.1 mini, total F1 increases from 0.758 with the best prompting baseline (Few-shot CoT) to 0.941, an improvement of 24.1%. The same trend holds for Claude Sonnet 4.5, where F1 increases from 0.794 to 0.870, and for DeepSeek V4-Flash, where F1 increases from 0.779 to 0.903. These correspond to relative improvements of 9.6% and 15.9%, respectively, indicating that the gain is not tied to a single backend model.

Figure 4 shows that prompting baselines degrade as intent complexity increases, especially at higher levels with multiple and implicit constraints. The Intent Engine remains stronger across levels, supporting the benefit of schema-bounded extraction and retrieval grounding for constructing complex orchestration-consumable SLO artifacts.

5.7.2. Structural Accuracy

We evaluate structural fidelity using Exact Match (Eq. 4) and Jaccard similarity (Eq. 5). Exact Match requires the entire predicted SLO set to match the ground truth, while Jaccard similarity measures constraint-set overlap.

Figure 5 shows that *Intent Engine* also achieves the strongest structural accuracy across models and prompting baselines. This indicates that the improvement is not limited to isolated constraint matches; the generated specifications more often preserve the complete SLO structure, avoid missing or spurious constraints, and remain consistent with the target schema.

Table 8
F1 Score comparison with the rule-based parser and prompting baselines using GPT-4.1 mini.

Method	Real	Synthetic	Total
Zero-shot	0.296	0.345	0.331
Zero-shot CoT	0.555	0.598	0.585
Rule-based Parser	**0.661**	**0.668**	**0.666**
Few-shot	0.711	0.768	0.751
Few-shot CoT	0.716	0.775	0.758
Intent Engine	**0.993**	**0.920**	**0.941**

5.7.3. Rule-based Parser Baseline

Table 8 compares the rule-based parser with the GPT-4.1 mini prompting baselines and *Intent Engine*. The rule-based parser achieves an overall F1 of 0.666, with similar performance on real and synthetic records. It outperforms zero-shot prompting, but remains below few-shot prompting and *Intent Engine*. This gap highlights the benefit of LLM-based intent interpretation and retrieval-grounded SLO construction for paraphrased and multi-constraint intents.

5.8. Failure Mode Analysis

Beyond measuring translation correctness on valid intents, we also evaluate how safely each method handles invalid intents. For such inputs, the desired behavior is not to generate a best-effort SLO, but to reject the request or return no orchestration-consumable SLO specification. We therefore analyze two aspects: (i) representative failure modes in intent-to-SLO translation, and (ii) rejection behavior across invalid categories.

Table 9 summarizes the main failure modes observed in the evaluation. The taxonomy distinguishes errors caused by

Figure 4: Level-wise F1 Scores across baselines and candidate models for real and synthetic records.

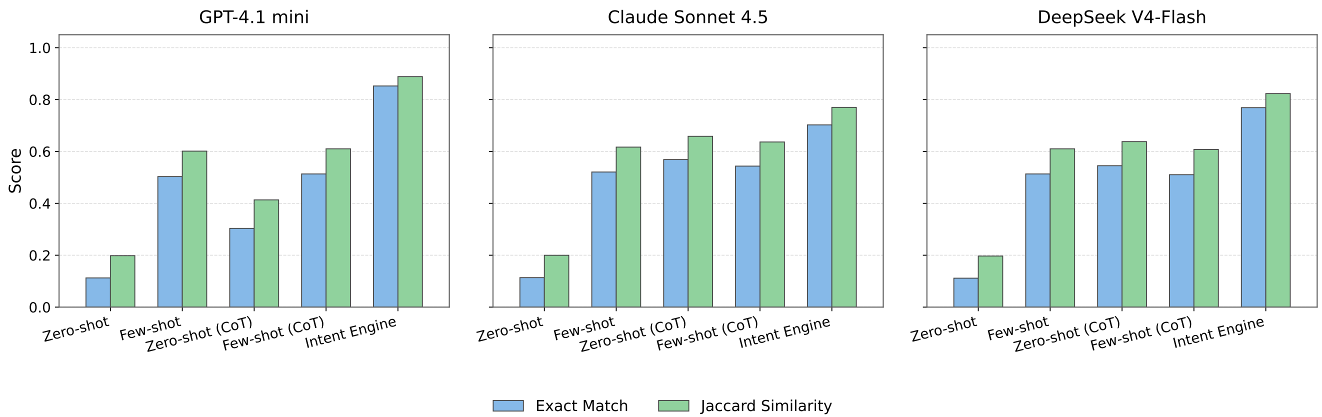

Figure 5: Exact Match and Jaccard similarity across models and translation baselines on the valid evaluation records.

unsupported or underspecified user requests, unconstrained generation, and missing or incorrect contextual grounding.

Using the rejection rate in Eq. 8, Figure 6 reports how often each method correctly avoids producing orchestration-consumable SLOs for invalid intents.

Figure 6 shows that the Intent Engine provides the most reliable failure handling across invalid intent categories. Overall, it correctly rejects 98.5% of invalid records, corresponding to only 3 false accepts out of 195 invalid intents. It achieves higher rejection on ambiguous, malformed, and unsupported intents, and reaches 93.9% rejection on conflicting intents.

The prompting baselines are less consistent: zero-shot variants reject many ambiguous and unsupported inputs,

Table 9
Failure modes in the intent-to-SLO translation process.

Failure mode	Example intent	Issue	System response
Unsupported	"Deploy the service with the lowest carbon footprint."	Requests a capability outside the supported SLO schema.	Reject as unsupported.
Incorrect value	"Place the service on the node with the lowest memory utilization."	Correct metric is detected, but the grounded value is missing or wrong.	Resolve through retrieval grounding.
Structural error	"Keep memory utilization reasonable."	Output contains malformed fields, units, or operators.	Reject during schema validation.
Ambiguous	"Place the service somewhere sensible with good performance."	Intent is underspecified and admits multiple interpretations.	Flag as ambiguous.
Conflicting	"Run the service on a node with highest memory utilization and lowest memory utilization."	Intent contains mutually inconsistent resource-preference constraints.	Reject as conflicting.
Grounding error	"Pick the node with the highest available storage."	Retrieved or selected contextual value is incorrect or stale.	Fallback or validation failure.

while few-shot variants often over-generate SLOs for conflicting intents. This suggests that examples improve structured output generation but do not reliably enforce safe rejection.

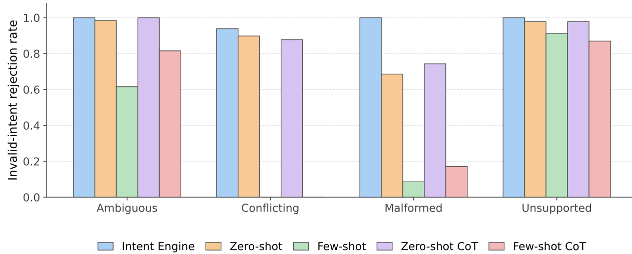

Figure 6: Invalid-intent rejection rate across invalid-intent categories using GPT-4.1 mini.

These results complement the hallucination analysis: hallucination metrics evaluate errors in generated SLOs, whereas rejection analysis measures whether invalid requests are blocked before validation. Together, they show that *Intent Engine* improves both valid-intent translation and invalid-intent rejection through schema-bounded extraction, retrieval grounding, and deterministic validation.

5.9. System Overhead

This section examines the overhead of *Intent Engine* as a translation layer for intent-driven orchestration (IDO) frameworks. The goal is to assess whether the architecture can produce SLO specifications within practical control-plane timescales and whether its context construction remains manageable as the monitored infrastructure grows. We therefore evaluate latency and prompt-context scalability for the intent-to-SLO translation phase.

5.9.1. Latency

We measure translation-layer latency from receiving a natural-language intent to producing the final validated SLO specification. Table 10 reports median, mean, and P95 latency on the total evaluation dataset for the LLM-backed *Intent Engine* configurations.

Table 10
Intent Engine runtime latency on the total evaluation dataset. Latency is measured from receiving a natural-language intent to producing the validated SLO specification.

Model	Median (s)	Mean (s)	P95 (s)
GPT-4.1 mini	2.35	2.60	4.30
Claude Sonnet 4.5	4.28	4.31	5.85
DeepSeek V4-Flash	7.78	8.76	17.43

The results show that *Intent Engine* operates at control-plane translation timescales. GPT-4.1 mini provides the lowest LLM-backed latency, with a median of 2.35 s and P95 of 4.30 s, while DeepSeek V4-Flash has the highest tail latency with a P95 of 17.43 s. For comparison, the rule-based parser completes symbolic parsing and snapshot resolution in sub-millisecond processing time, but with lower F1 than *Intent Engine*. This reflects the expected accuracy–generality trade-off between deterministic parsing and LLM-based intent interpretation. Since intent translation is not performed on the data path of individual service requests, these latencies are suitable for human-triggered or low-frequency orchestration updates where the generated SLOs are consumed by downstream placement or orchestration algorithms.

5.9.2. Scalability

We evaluate scalability using the LLM-visible context size required for intent-to-SLO translation as the monitored infrastructure grows. The six-node testbed snapshot is scaled to 1000 nodes while preserving the same node-state schema. We compare full snapshot prompting, which inserts the entire continuum state into the prompt, with *Intent Engine*'s retrieval-grounded context construction, which inserts only

Table 11
Prompt-context size with increasing infrastructure scale.

Nodes	Full prompt	Retrieved context
6	3,242	1,104
25	12,694	2,012
50	25,255	2,011
100	50,368	2,012
250	125,600	2,012
500	250,947	2,008
1000	501,726	2,008

the top-k metric-relevant evidence chunks. We use $k = 8$, matching the evaluated configuration.

Token counts are measured with the GPT-4.1 mini tokenizer. Absolute counts may vary across models, but the architectural trend remains the same: full snapshot prompting grows with infrastructure size, whereas top-k retrieved context is bounded by the retrieval budget.

Table 11 shows that full snapshot prompting increases from 3,242 tokens at 6 nodes to 501,726 tokens at 1000 nodes. In contrast, retrieved context stabilizes near 2,000 tokens after 25 nodes because the query retrieves the full $k = 8$ evidence budget. For multiple implicit SLOs, retrieved context grows with the number of grounding queries and is bounded by $m \times k$ chunks, where m is the number of implicit grounded constraints. These results show that *Intent Engine* limits LLM-visible context growth while preserving access to infrastructure state for grounding.

5.10. Downstream Placement Impact

To examine whether SLO translation errors affect downstream placement behavior, we perform a placement-impact analysis using the SLO-driven node-matching algorithm from MicroIntent [19] placement generator. *Intent Engine* remains a platform-agnostic SLO construction layer; the placement generator is used only as a deterministic evaluator to measure how translated SLOs influence a downstream placement decision and accuracy of intended placement intent.

For each valid intent, the placement generator is first applied to the ground-truth SLO label and the corresponding compute-continuum snapshot to check whether a feasible placement exists. The same generator is then applied to each method's predicted SLOs. The generator filters candidate nodes by placement layer, resource constraints, and network constraints, and returns the first node satisfying all SLOs. Placement impact is computed only on the 399 valid records whose ground-truth SLO labels have at least one feasible placement in the snapshot.

Table 12 shows that prompting-based translation errors mainly propagate as no-match placement outcomes. The best prompting baseline has a 30.8% placement failure rate, while the Intent Engine reduces this to 2.1%. The largest gain comes from reducing no-match cases from 29.3–30.1% to 0.8%. Invalid placements remain low because the deterministic placement generator is conservative: incomplete

Table 12
Downstream placement failure using GPT-4.1 mini.

Method	No match	Invalid placement	Failure
Zero-shot	30.1%	1.5%	31.6%
Few-shot	29.3%	1.8%	31.1%
Zero-shot CoT	30.1%	1.5%	31.6%
Few-shot CoT	29.3%	1.5%	30.8%
Intent Engine	**0.8%**	**1.3%**	**2.1%**

or inconsistent SLOs usually produce no matching node rather than a returned node that violates the ground-truth constraints. These results show that reliable SLO artifact construction reduces the risk of intent-acquisition errors propagating into downstream placement decisions.

6. Limitations

We designed *Intent Engine* as an intent acquisition and SLO construction layer, not a fully autonomous IDO framework. *Intent Engine* validates and grounds the SLO artifact, while downstream IDO frameworks remain responsible for placement, deployment execution, enforcement, re-grounding, and runtime assurance. For system-critical placement decisions, the generated SLO should be inspected before execution, since stale context, malformed intents, or incorrectly accepted constraints may lead to service disruption.

This work focuses on natural-language-to-SLO construction for compute-continuum service placement. Autoscaling, fault recovery, migration, cost optimization, and closed-loop assurance are outside the current evaluation. The supported intents are also mainly infrastructure-oriented, covering placement, compute capacity, utilization, storage, and network constraints; business-level goals, privacy, energy, carbon-awareness, application dependencies, and multi-service workflow constraints are not included in the present schema.

The grounding mechanism relies on monitored infrastructure snapshots. Although this resolves implicit SLO values from real system state, the values reflect conditions at snapshot time and may become stale in highly dynamic environments. Thus, the generated SLO is a snapshot-grounded specification rather than an always-current runtime guarantee, while re-grounding after initial placement remains part of the downstream IDO intent assurance phase.

The evaluation uses a six-node edge–cloud testbed and the TeaStore reference application to construct grounded intent-to-SLO records. TeaStore is only a reference microservice application, and the pipeline remains application-agnostic. However, the current setup cannot fully evaluate node-scale behavior or retrieval-augmented context selection in large continuum systems. To the best of our knowledge, no richer public grounded intent-to-SLO dataset exists for this task; once available, *Intent Engine* can be evaluated across larger infrastructures, broader heterogeneity, and more complex deployments.

Finally, *Intent Engine* relies on LLMs for semantic extraction and grounded SLO generation, which can still produce errors despite schema constraints and validation. The invalid-intent rejection and hallucination analyses show that *Intent Engine* reduces unsafe outputs, but does not eliminate all value-level errors. Since no contextual dataset currently exists, training or fine-tuning a domain-specific LLM for accurate SLO construction is not yet feasible.

7. Conclusion and Future Work

This paper presented *Intent Engine*, a natural-language-to-SLO construction architecture for compute-continuum service placement. It serves as an intent acquisition layer that transforms unstructured placement intents into validated, orchestration-consumable SLO artifacts for downstream IDO and placement frameworks.

The architecture combines schema-bounded intent extraction, retrieval-grounded value construction, and schema validation to separate natural-language interpretation from downstream orchestration decisions. Evaluation on a real edge–cloud testbed dataset shows that *Intent Engine* improves constraint-level correctness, structural accuracy, hallucination reduction, implicit-value grounding, and invalid-intent rejection compared with prompt-only LLM baselines and a non-LLM rule-based parser. The results show that reliable intent-to-SLO construction requires infrastructure-aware grounding and schema-constrained validation, especially for implicit, multi-constraint, and invalid intents.

Overall, this work shows that natural-language interfaces for compute-continuum orchestration should treat generated SLOs as control-plane artifacts rather than free-form text outputs. By preserving the boundary between intent interpretation and downstream placement execution, *Intent Engine* provides a practical path for integrating natural-language intent acquisition with existing IDO frameworks while leaving placement optimization, deployment actuation, and runtime QoS assurance to the consuming orchestration system.

Future work will investigate intent drift and closed-loop intent assurance after constructed SLOs are consumed by orchestration systems. We plan to extend *Intent Engine* toward a full IDO framework that ingests generated SLO artifacts and verifies runtime intent satisfaction through re-grounding, feasibility checking, conflict detection and resolution, and intent negotiation after initial service placement.

CRediT authorship contribution statement

Koushikur Islam: Conceptualization, Investigation, Methodology, Data Curation, Software, Visualization, Writing - Original Draft. **Rodrigo N. Calheiros:** Conceptualization, Methodology, Writing - Review & Editing.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this manuscript, the authors used OpenAI's ChatGPT for grammar checking and language refinement. All content was reviewed and edited by the authors, who take full responsibility for the accuracy and integrity of the final manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have influenced the work reported in this paper.

Acknowledgment

This work did not receive any grant from funding agencies in the public, commercial, or non-profit sectors.

Data availability

The dataset used in this study is publicly available to support reproducibility and further research. The public version of the dataset can be accessed via Zenodo at:
https://doi.org/10.5281/zenodo.20810799

A. Prompt templates

A.1. Intermediate SLO parser prompt template

See Figure 7.

A.2. Zero-Shot and Zero-Shot CoT prompt template

See Figure 8.

A.3. Few-Shot and Few-Shot CoT prompt template

See Figure 9.

B. Dataset samples

B.1. Dataset sample

See Figure 10.

References

[1] M. Sebrechts, B. Volckaert, F. De Turck, K. Yang, M. Al-Naday, Fog native architecture: Intent-based workflows to take cloud native toward the edge, IEEE Communications Magazine 60 (2022) 44–50.

[2] I. Odun-Ayo, M. Ananya, F. Agono, R. Goddy-Worlu, Cloud computing architecture: A critical analysis, in: 2018 18th international conference on computational science and applications (ICCSA), IEEE, 2018, pp. 1–7.

[3] Y. Mansouri, M. A. Babar, A review of edge computing: Features and resource virtualization, Journal of Parallel and Distributed Computing 150 (2021) 155–183.

[4] S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hästbacka, D. Taibi, Cloud continuum: The definition, IEEE Access 10 (2022) 131876–131886.

```
SYSTEM_PROMPT = f"""
You are an information extractor. Convert a natural-language service intent into a JSON SLO schema.

CRITICAL RULES:
- Output MUST be valid JSON only. No markdown.
- Do NOT invent numbers. If no explicit numeric threshold/range exists, keep goal.value=null and goal.
- If expression uses qualitative terms ("high", "low", "stable", "minimal", "fast"), set qualifier accordingly.
- Use op as:
  - "compare" when user gives explicit threshold/range (<=, >=, between, equals)
  - "aggregate" when user asks average/mean/median/min/max without naming a node/link explicitly
  - "rank" when user asks highest/lowest or uses "most"/"least"
- If user says "higher/lower" without numbers, treat as rank with qualifier where possible.

PLACEMENT (VERY STRICT):
- ONLY set placement_location IF the expression explicitly specifies a deployment location.
  Explicit location cues include: "edge", "cloud", "in the cloud", "on the edge", "edge layer", "cloud layer",
  "deploy on edge", "deploy in cloud", "run on edge", "place in cloud", "placement location".
- If NO explicit location cue exists:
  - Set layer="any"
  - Set placement_location.target.op = null
  - Set placement_location.target.value = null

DEFAULT UNITS (when not specified):
- cpu_capacity: "cores"
- memory_capacity/storage_capacity: "GB"
- utilizations: "%"
- bandwidth: "Mbps"
- port_utilization: "%"

OUTPUT SHAPE:
Return a single JSON object with fields:
{
  "service": string,
  "layer": "edge"|"cloud"|"both"|"any",
  "intents": [
    {"type":"placement_location","target":{"op":"set"|null,"value":"edge|cloud|hybrid"|null}},
    {"type":"static_compute_capacity","target":[TargetItem...]},
    {"type":"dynamic_compute_utilization","target":[TargetItem...]},
    {"type":"single_point_network_resource","target":[TargetItem...]},
  ]
}

TargetItem:
{
  "metric": string,
  "unit": string,
  "op": "compare"|"aggregate"|"rank",
  "goal": {"cmp":"<="|">="|"between"|"="|null, "value": number|null},
  "aggregate": "min"|"max"|"avg"|"mean"|"median"|null,
  "rank": "highest"|"lowest"|null,
  "qualifier": "high"|"medium"|"low"|"stable"|"minimal"|"fast"|null
}
""".strip()

USER_PROMPT = f"""
Input:
{{
  "service": "{{service}}",
  "intent_expression": "{{expression}}",
  "schema_template": {{
    "service": "",
    "placement_location": "any",
    "intents": [
      {{"type": "placement_location", "target": {{"op": null, "value": null}}}},
      {{"type": "static_compute_capacity", "target": []}},
      {{"type": "dynamic_compute_utilization", "target": []}},
      {{"type": "single_point_network_resource", "target": []}}
    ]
  }},
  "allowed_enums": {{
    "layer": ["edge","cloud","both","any"],
    "placement_op": ["set", null],
    "placement_value": ["edge","cloud","both", null],
    "op": ["compare","aggregate","rank"],
    "cmp": ["<=",">=","between","=", null],
    "aggregate": ["min","max","avg","mean","median", null],
    "rank": ["highest","lowest", null],
    "qualifier": ["high","medium","low","stable","minimal","fast", null]
  }}

Answer:
""".strip()
```

Figure 7: The prompt template used for intent expression decomposition to construct intermediate SLO specification with explicit requirements.

```
SYSTEM_PROMPT = f'''
Instruction:
Service: {{service}}
Intent Expression: {{expression}}
Snapshot (JSON): {{snapshot}}

You are an information extraction system that converts natural-language service placement intents into SLO labels.

IMPORTANT:
Output MUST be JSON only.

Return ONLY valid JSON matching:
{
  "service": "<service>",
  "slos": [
    {
      "metric": <string or null>,
      "target": {
        "cmp": "<=" | ">=" | "between" | "=" | null,
        "op": "highest" | "lowest" | "set" | null,
        "value": <number | string | null>
      },
      "type": "placement_location" | "static_compute_resource" | "dynamic_compute_resource" | "node_network_resource",
      "unit": <string or null>
    }
  ]
}

Constraints:
1) Use only these types/metrics:
- static_compute_resource: cpu_capacity, memory_capacity, storage_capacity
- dynamic_compute_resource: cpu_utilization, memory_utilization, storage_utilization
- node_network_resource: bandwidth, port_utilization
- placement_location: metric=null, unit=null

2) Use canonical units for output:
cpu_capacity=cores;
memory_capacity=bytes;
storage_capacity=bytes;
*_utilization=percent;
bandwidth=Mbps;
port_utilization=percent;

Procedure:
- Identify each constraint in the expression.
- Parse numeric values and their units (if any).
- Normalize numeric values into the canonical unit for that metric using standard unit conventions (SI/IEC for data sizes; time and rate unit conversions as
standard).
- Emit the JSON.

Snapshot use:
- Only consult the snapshot when the expression requests an extremum (highest/lowest/etc.).
- When comparing snapshot values, normalize them to canonical units first, then choose max/min.
- Do not invent numeric values.

Placement:
- Only emit placement_location if edge/cloud/both/any is explicitly stated;
  encode with target.op="set" and target.value.
'''.strip()


USER_PROMPT = f"""
Input:
{{
  "service": "{service}",
  "expression": "{expression}",
  "snapshot": {snapshot}
}}

Answer:
"""
```

Figure 8: The prompt template used for Zero-Shot learning. For Zero-Shot CoT, we put the phrase "Let's think step-by-step" in the prompt template 8.

[5] Z. N. Samani, N. Mehran, D. Kimovski, R. Prodan, Proactive sla-aware application placement in the computing continuum, in: 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2023, pp. 468–479.

[6] A. Zafeiropoulos, E. Fotopoulou, C. Vassilakis, I. Tzanettis, C. Lombardo, A. Carrega, R. Bruschi, Intent-driven distributed applications management over compute and network resources in the computing continuum, in: 2023 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT), IEEE, 2023, pp. 429–436.

[7] M. Mota-Cruz, J. H. Santos, J. F. Macedo, K. Velasquez, D. P. Abreu, Optimizing microservices placement in the cloud-to-edge continuum: A comparative analysis of app and service based approaches, in: 2024 IEEE 22nd Mediterranean Electrotechnical Conference (MELECON), IEEE, 2024, pp. 1321–1326.

```
SYSTEM_PROMPT = f'''
Instruction:
Service: {{service}}
Intent Expression: {{expression}}
Snapshot (JSON): {{snapshot}}

You are an information extraction system that converts natural-language service intents into SLO labels.......... (Same system prompt as Zero-Shot)

USER_PROMPT = f"""
Example 1
Input:
{{
  "service": "svc/example-auth",
  "expression": "Deploy svc/example-auth, keep CPU utilization below 55 percent.",
  "snapshot": {{"nodes": [{{"name": "n1", "cpu_utilization": 40}}, {{"name": "n2", "cpu_utilization": 62}}]}}
}}
Output:
{{
  "service": "svc/example-auth",
  "slos": [
    {{
      "metric": "cpu_utilization",
      "target": {{"cmp": "<=", "op": null, "value": 55, "range": null}},
      "type": "dynamic_compute_resource",
      "unit": "percent"
    }}
  ]
}}

Example 2
Input:
{{
  "service": "svc/example-cache",
  "expression": "Run svc/example-cache with at least 2 GB memory.",
  "snapshot": {{"nodes": [{{"name": "n1", "memory_capacity": 1073741824}}, {{"name": "n2", "memory_capacity": 8589934592}}]}}
}}
Output:
{{
  "service": "svc/example-cache",
  "slos": [
    {{
      "metric": "memory_capacity",
      "target": {{"cmp": ">=", "op": null, "value": 2000000000, "range": null}},
      "type": "static_compute_resource",
      "unit": "bytes"
    }}
  ]
}}

Example 3
Input:
{{
  "service": "svc/example-worker",
  "expression": "Deploy svc/example-worker with at least 500 MiB of storage capacity.",
  "snapshot": {{"nodes": [{{"name": "n1", "storage_capacity": 800000000}}, {{"name": "n2", "storage_capacity": 600000000}}]}}
}}
Output:
{{
  "service": "svc/example-worker",
  "slos": [
    {{
      "metric": "storage_capacity",
      "target": {{"cmp": ">=", "op": null, "value": 524288000, "range": null}},
      "type": "static_compute_resource",
      "unit": "bytes"
    }}
  ]
}}

(3 more examples....)

Input:
{{
  "service": "{service}",
  "expression": "{expression}",
  "snapshot": {snapshot}
}}

Answer:
"""
```

Figure 9: The prompt template used for Few-Shot learning. For Few-Shot CoT, we put the phrase "Let's think step-by-step" is added along with Zero-Shot prompt template 8 and provided with total 6 examples.

Example 1: Valid explicit constraint (Level 1)

{
  "input": {
    "service": "descartesresearch/teastore-image",
    "expression": "For descartesresearch/teastore-image, please keep CPU utilization below 65%."
  },
  "label": {
  "slos": [{"type": "dynamic_compute_resource", "metric": "cpu_utilization", "target": {"cmp": "≤", "value": 65}, "unit": "percent"}]
  },
  "meta": {"level": 1, "source": "real", "status": "valid"}
}

Example 2: Valid placement and numeric constraints (Level 2)

{
  "input": {
    "service": "descartesresearch/teastore-auth",
    "expression": "Deploy descartesresearch/teastore-auth at the edge and keep storage utilization below 45%."
  },
  "label": {
  "slos": [{"type": "placement_location", "target": {"op": "set", "value": "edge"}}, {"type": "dynamic_compute_resource", "metric": "storage_utilization", "target": {"cmp": "≤", "value": 45}, "unit": "percent"}]
  },
  "meta": {"level": 2, "source": "real", "status": "valid"}
}

Example 3: Valid implicit constraint (Level 4)

{
  "input": {
    "service": "descartesresearch/teastore-recommender",
    "expression": "Place the service on the edge layer and pick the node with the lowest current memory utilization."
  },
  "label": {
  "slos": [{"type": "placement_location", "target": {"op": "set", "value": "edge"}}, {"type": "dynamic_compute_resource", "metric": "memory_utilization", "target": {"cmp": "≤", "op": "lowest", "value": 34.271}, "unit": "percent"}]
  },
  "meta": {"level": 4, "source": "real", "status": "valid"}
}

Example 4: Non-valid ambiguous intent

{
  "input": {
    "service": "descartesresearch/teastore-db",
    "expression": "Deploy descartesresearch/teastore-db somewhere sensible, with enough headroom and generally good performance."
  },
  "label": {"slos": []},
  "meta": {"level": 3, "source": "real", "status": "ambiguous"}
}

Example 5: Non-valid unsupported intent

{
  "input": {
    "service": "descartesresearch/teastore-db",
    "expression": "Deploy descartesresearch/teastore-db with the lowest carbon footprint and GDPR-compliant storage."
  },
  "label": {"slos": []},
  "meta": {"level": 3, "source": "real", "status": "unsupported"}
}

Figure 10: Dataset record examples across valid and invalid natural language intents and ground-truth SLO labels.

[8] A. Boutouchent, A. N. Meridja, Y. Kardjadja, A. M. Maia, Y. Ghamri-Doudane, M. Koudil, R. H. Glitho, H. Elbiaze, Amanos: An intent-driven management and orchestration system for next-generation cloud-native networks, IEEE Communications Magazine 62 (2023) 42–49.
[9] M. Gharbaoui, B. Martini, P. Castoldi, Intent-based networking: Current advances, open challenges, and future directions, in: 2023 23rd International Conference on Transparent Optical Networks (ICTON), IEEE, 2023, pp. 1–5.
[10] A. Clemm, L. Ciavaglia, L. Z. Granville, J. Tantsura, Intent-based networking-concepts and definitions (2022).
[11] J. Spillner, J. F. Borin, L. F. Bittencourt, Intent-based placement of microservices in computing continuums, in: Future Intent-Based Networking: On the QoS Robust and Energy Efficient Heterogeneous Software Defined Networks, Springer, 2021, pp. 38–50.
[12] A. Morichetta, N. Spring, P. Raith, S. Dustdar, Intent-based management for the distributed computing continuum, in: 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE), IEEE, 2023, pp. 239–249.
[13] N. Filinis, I. Tzanettis, D. Spatharakis, E. Fotopoulou, I. Dimolitsas, A. Zafeiropoulos, C. Vassilakis, S. Papavassiliou, Intent-driven orchestration of serverless applications in the computing continuum, Future Generation Computer Systems 154 (2024) 72–86.
[14] N. Akbari, A. N. Toosi, J. Grundy, H. Khalajzadeh, M. S. Aslanpour, S. Ilager, Icontinuum: An emulation toolkit for intent-based computing across the edge-to-cloud continuum, in: 2024 IEEE 17th International Conference on Cloud Computing (CLOUD), IEEE, 2024, pp. 468–474.
[15] T. Metsch, M. Viktorsson, A. Hoban, M. Vitali, R. Iyer, E. Elmroth, Intent-driven orchestration: Enforcing service level objectives for cloud native deployments, SN Computer Science 4 (2023) 268.
[16] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, D. Roth, Recent advances in natural language processing via large pre-trained language models: A survey, ACM Computing Surveys 56 (2023) 1–40.
[17] Z. He, T. Liang, W. Jiao, Z. Zhang, Y. Yang, R. Wang, Z. Tu, S. Shi, X. Wang, Exploring human-like translation strategy with large language models, Transactions of the Association for Computational Linguistics 12 (2024) 229–246.
[18] Y. Bang, Z. Ji, A. Schelten, A. Hartshorn, T. Fowler, C. Zhang, N. Cancedda, P. Fung, Hallulens: Llm hallucination benchmark, arXiv preprint arXiv:2504.17550 (2025).
[19] K. Islam, G. D. C. Rodrigues, B. Javadi, R. N. Calheiros, Microintent: Intent-based placement strategy for microservice application in the compute continuum using llms, in: 2025 IEEE International Conference on Smart Internet of Things (SmartIoT), IEEE, 2025, pp. 124–131.
[20] J. Santos, T. Wauters, B. Volckaert, F. De Turck, Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions, IEEE Communications Surveys & Tutorials 23 (2021) 2557–2589.
[21] M. Asif, T. A. Khan, W.-C. Song, Leveraging cognitive machine reasoning and nlp for automated intent-based networking and e2e service orchestration, IEEE Access (2025).
[22] Kubernetes, Kubernetes documentation, https://kubernetes.io/docs/, 2026. Accessed: January 15, 2026.
[23] Y. Xiong, Y. Sun, L. Xing, Y. Huang, Extend cloud to edge with kubeedge, in: 2018 IEEE/ACM Symposium On Edge Computing (SEC), IEEE, 2018, pp. 373–377.
[24] M. Jansen, L. Wagner, A. Trivedi, A. Iosup, Continuum: Automate infrastructure deployment and benchmarking in the compute continuum, in: Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, 2023, pp. 181–188.
[25] A. Al-Dulaimy, M. Jansen, B. Johansson, A. Trivedi, A. Iosup, M. Ashjaei, A. Galletta, D. Kimovski, R. Prodan, K. Tserpes, et al., The computing continuum: From iot to the cloud, Internet of Things 27 (2024) 101272.
[26] A. Morichetta, J. Brenes, M. Kolobov, D. Dib, T. Metsch, A. Lackinger, C. Capova, H. Song, R. Dautov, A. Khalid, et al., incoord: Intent-based coordination in the multi-domain cloud-edge continuum, in: 2025 IEEE 33rd International Conference on Network Protocols (ICNP), IEEE, 2025, pp. 1–6.
[27] B. Sedlak, V. C. Pujol, P. K. Donta, S. Dustdar, Diffusing high-level slo in microservice pipelines, in: 2024 IEEE International Conference on Service-Oriented System Engineering (SOSE), IEEE, 2024, pp. 11–19.
[28] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, S. G. Rao, Hey, lumi! using natural language for {intent-based} network management, in: 2021 usenix annual technical conference (usenix atc 21), 2021, pp. 625–639.
[29] C. Capova, A. Morichetta, A. Lackinger, S. Dustdar, Intent-to-learning translation for computing continuum management, in: 2025 IEEE 33rd International Conference on Network Protocols (ICNP), IEEE, 2025, pp. 1–6.
[30] A. Esashi, P. Lertpongrujikorn, S. Kato, M. A. Salehi, Action engine: Automatic workflow generation in faas, Future Generation Computer Systems 174 (2026) 107947.
[31] M. Asif, T. A. Khan, W.-C. Song, Evaluating large language models for optimized intent translation and contradiction detection using knn in ibn, IEEE Access (2025).
[32] A. Mekrache, A. Ksentini, Llm-enabled intent-driven service configuration for next generation networks, in: 2024 IEEE 10th International Conference on Network Softwarization (NetSoft), IEEE, 2024, pp. 253–257.
[33] A. Mekrache, A. Ksentini, C. Verikoukis, Intent-based management of next-generation networks: An llm-centric approach, Ieee Network 38 (2024) 29–36.
[34] A. Mekrache, A. Ksentini, C. Verikoukis, Oss-gpt: An llm-powered intent-driven operations support system for 6g networks, in: 2025 IEEE 11th International Conference on Network Softwarization (NetSoft), IEEE, 2025, pp. 155–163.
[35] A. Mekrache, A. Ksentini, C. Verikoukis, Next-generation 6g network management with oss-gpt, in: Proceedings of the ACM SIGCOMM 2025 Posters and Demos, 2025, pp. 158–160.
[36] A. Mekrache, A. Ksentini, C. Verikoukis, Dmo-gpt: An intent-driven framework for distributed 6g management and orchestration, IEEE Communications Magazine (2025).
[37] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, Y. Iwasawa, Large language models are zero-shot reasoners, Advances in neural information processing systems 35 (2022) 22199–22213.
[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al., Chain-of-thought prompting elicits reasoning in large language models, Advances in neural information processing systems 35 (2022) 24824–24837.
[39] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al., Retrieval-augmented generation for knowledge-intensive nlp tasks, Advances in neural information processing systems 33 (2020) 9459–9474.
[40] Y. Zhang, H. Cai, X. Song, Y. Chen, R. Sun, J. Zheng, Reverse chain: A generic-rule for llms to master multi-api planning, in: Findings of the Association for Computational Linguistics: NAACL 2024, 2024, pp. 302–325.
[41] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, Y. Cao, React: Synergizing reasoning and acting in language models, in: The eleventh international conference on learning representations, 2022.
[42] P. Verma, S. P. Midigeshi, G. Sinha, A. Solin, N. Natarajan, A. Sharma, Plan-rag: Planning-guided retrieval augmented generation (2024).
[43] Amazon Web Services, Aws ec2 documentation, https://docs.aws.amazon.com/ec2/, 2026. Accessed: January 15, 2026.
[44] Tailscale, Tailscale documentation, https://tailscale.com/kb, 2026. Accessed: January 15, 2026.
[45] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, S. Kounev, Teastore: A micro-service reference application for benchmarking, modeling and resource management research, in: 2018

IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2018.
[46] OpenAI, Openai documentation, https://platform.openai.com/docs/overview, 2026. Accessed: January 15, 2026.
[47] Flask, Flask documentation, https://flask.palletsprojects.com/en/stable/, 2026. Accessed: January 15, 2026.
[48] Prometheus, Prometheus documentation, https://prometheus.io/docs/introduction/overview/, 2026. Accessed: January 15, 2026.
[49] Grafana, Grafana documentation, https://grafana.com/docs/, 2026. Accessed: January 15, 2026.
[50] Amazon Web Services, Aws s3 documentation, https://docs.aws.amazon.com/s3/, 2026. Accessed: January 15, 2026.
[51] Qdrant, Qdrant documentation, https://qdrant.tech/documentation/, 2026. Accessed: January 15, 2026.

Koushikur Islam is a Research Assistant at the Smart and Distributed Computing Lab at Western Sydney University, Australia. His research focuses on improving resource management across the edge–cloud continuum through adaptive and autonomous AI/ML-driven solutions. He received his Master of Information and Communications Technology degree from Western Sydney University with High Distinction. He also holds a Bachelor of Science in Computer Science and Engineering from the American International University–Bangladesh, graduating with the Summa Cum Laude (Gold Medal) distinction.

Dr Rodrigo N. Calheiros is an Associate Professor in the School of Computer, Data and Mathematical Sciences, Western Sydney University, Australia. He conducts applied research in diverse aspects of distributed computing systems, including cloud computing, edge computing, and Internet of Things. He co-authored more than 100 papers, which attracted together 21,000 Google Scholar citations. He is a Fellow of Advance HE, Senior Member of the IEEE and Senior Member of the ACM.