ACM 4th International Conference on
Information Technology for Social Good (GoodIT 2024)

CuLao - Constructing Utilities of Large Language Models in Resource-Constrained Environments

Hong-Linh Truong
Department of Computer Science, Aalto University
linh.truong@aalto.fi

Nguyen Ngoc Nhu Trang
Daienso Lab
nhutrang.nguyen@daienso.com

MOTIVATION

The increasing development and utilization of Large Language Model (LLM) services have demonstrated many benefits in different contexts. However, LLM services are mainly available in the public cloud and require huge computing resources to operate, thus not accessible to many **Companies, Organizations, and Communities with cOnstrained reSources (COCOS)**, where there is a lack of networks, machines and ML/LLM expert capabilities.

CuLao: a framework for constructing utilities from LLMs in **resource-constrained computing environments** (**RCEs**), focuses on key requirements of **COCOS** by enabling the provisioning and coordination of LLMs utilities, based on edge LLMs.

UNDERSTANDING CONTEXTS AND REQUIREMENTS

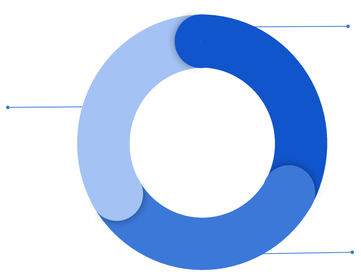
Provisioning LLMs in COCOS are driven by three different main contexts:

- **ML infrastructure context**: related to network and computing resources required by LLMs.
- **Application purpose context**: related to the goal of using LLMs, helping to select LLMs features and identify LLMs issues.
- **Operational context**: related to costs, security and policy issues, and energy and sustainability conditions that impact on the operation.

Requirements (RQs) related to infrastructures and operational contexts

RQ1

Provisioning LLM utilities in resource-constrained computing environments of non-dedicated desktops, medium servers and edge devices



RQ2

Sharing LLM utilities within and between resource-constrained computing environments

RQ3

Easing the development tasks for integration with suitable COCOS applications

ARCHITECTURE, MODELS AND TECHNIQUES

Making LLMs as service utilities

- ***Encapsulating LLMs into services***: *AMQP-based* LLM utilities as they fulfill the need of asynchronous invocation, easy to manage requests. To manage a LUE, each LUE has a catalog specifying detailed information about LLMs, executors and required parameters.
- ***Interfacing with individual LLMs***: two layers of queries handling in RCEs are included: LLM utility itself (e.g., for AMQP) and *LLM-specific Gateways/LUE Coordinator*.

```python
# create a LLMQueue as API for the service
llmsinglequeue=LLMSingleQueue(service_config["params"])
# prepare the model instance based on the service name
llm_model=COLLMS_MODELS[service_name](service_config)
llmsinglequeue.set_model(llm_model)
#announce the service as an utility
sd=ServiceDiscovery()
if service_info is not None:
    sd.publish_service(service_name,service_info)
# serving
llmsinglequeue.serving()
```

Code excerpt illustrating how to load a model based on its configuration to instantiate an AMQP-based LLM utility.

```yaml
coedit:
  model_path: llms/coedit-large
  executor: collms
  params:
    model_params:
      max_length: 512
    amqp:
      url:
      query_queue_name: coeditquery
      answer_queue_name: coeditanswer
    max_tokens: 256
```

Simplified example of an entry in a LUE Catalog.

Discovering and sharing LLM utilities

Utilities can be started and stopped arbitrarily due to non-dedicated resources. We use *Consul* for service discovery and secret management. Based on the configuration, the *LUE Coordinator* can share LLM service information to another LUE.

```json
"service_name": "openhermes",
"id": "e8e17fb2-bc92-4513-b2a9-81766cd01efc",
"tag_list": ["llm", "openhermes"],
"lue_id": "culao01",
"protocol": "AMQP",
"service_config": {
    "params": {
        "amqp": {
            "url": "amqps://...",
            "query_queue_name": "openhermesquery",
            "answer_queue_name": "openhermesanswer",
            "answer_fanout": false
        }
    }
}
```

Example of simplified service information.

Routing and coordinating queries and answers

```yaml
taskmodel:
  qa:
    - phi3
  text2text:
    - openhermes
  grammar:
    - coedit
```

A simple, easy-to-specify configuration.

To route queries to the right LLM utilities, we develop a basic task model encapsulating key information:

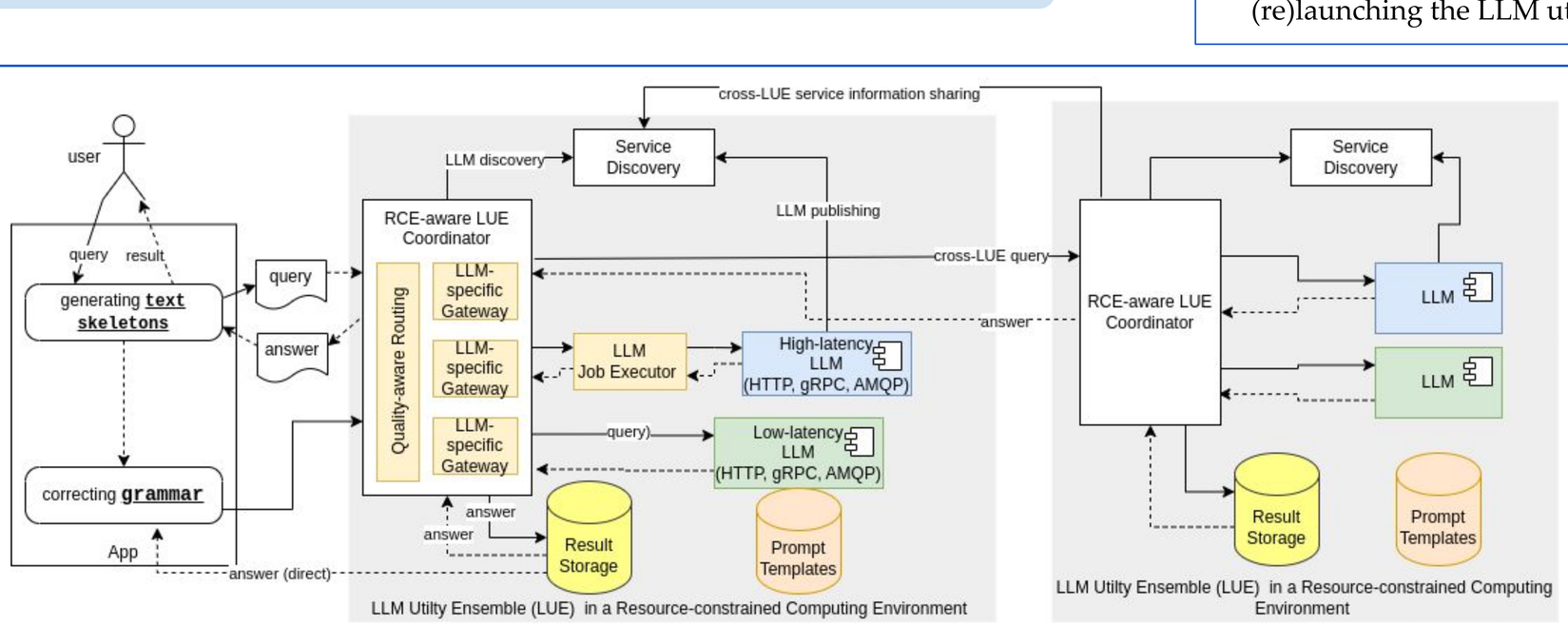
- *task type*: represents the selected type of tasks in LUE
- *service tags*: tags used identifying LLM utilities
- *execution mode*: represents information about the execution of requests

Handling results for applications

Results are buffered in *Result Storage*. Message chats, generated code and generated texts are stored in *Redis*. Results from LLMs are always stored into Result Storage before being pulled back to the application. This is done by a combination work of *LLM-specific Gateway* and the *Coordinator*.

Updating LLMs and integrating new LLMs

(i) assessing new versions of LLMs or new LLMs, (ii) pulling LLM models and building LLM utilities, (iii) providing configuration information, (iv) (re)launching the LLM utilities



High-level view of CuLao: harmonizing LLMs as utilities in resource constrained computing environments.

Within an RCE, CuLao establishes a LLM Utility Ensemble (LUE), which includes:

- ***LLMs***: are selected and optimized based on COCOS needs.
- ***Service Discovery***: monitors and publishes available LLM utilities.
- ***RCE-aware LUE Coordinator***: decides suitable LLM utilities and route queries.
- ***Result Storage***: for asynchronous retrievals.
- ***Prompt Templates***: support building queries.