Couler: Unified Machine Learning Workflow Optimization in Cloud

Xiaoda Wang[§], Yuan Tang[†], Tengda Guo[§], Bo Sang[*]
Jingji Wu[*], Jian Sha[*], Ke Zhang[*], Jiang Qian[‖], Mingjie Tang[§]
[*]*Ant Group* [†]*Akuity, Inc.* [‖]*Snap, Inc* [§]*Sichuan University*
{b.sang, jingji.wjw, shajian, yingzi.zk}@antgroup.com,
{wangxiaoda, guotengda}@stu.scu.edu.cn, {terrytangyuan, tangrock}@gmail.com

***Abstract*—Machine Learning (ML) has become ubiquitous, fueling data-driven applications across various organizations. Contrary to the traditional perception of ML in research, ML workflows can be complex, resource-intensive, and time-consuming. Expanding an ML workflow to encompass a wider range of data infrastructure and data types may lead to larger workloads and increased deployment costs. Currently, numerous workflow engines are available (with over ten being widely recognized). This variety poses a challenge for end-users in terms of mastering different engine APIs. While efforts have primarily focused on optimizing ML Operations (MLOps) for a specific workflow engine, current methods largely overlook workflow optimization across different engines.**

In this work, we design and implement COULER , a system designed for unified ML workflow optimization in the cloud. Our main insight lies in the ability to generate an ML workflow using natural language (NL) descriptions. We integrate Large Language Models (LLMs) into workflow generation, and provide a unified programming interface for various workflow engines. This approach alleviates the need to understand various workflow engines' APIs. Moreover, COULER enhances workflow computation efficiency by introducing automated caching at multiple stages, enabling large workflow auto-parallelization and automatic hyperparameters tuning. These enhancements minimize redundant computational costs and improve fault tolerance during deep learning workflow training. COULER is extensively deployed in real-world production scenarios at ANT GROUP , handling approximately 22k workflows daily, and has successfully improved the CPU/Memory utilization by more than 15% and the workflow completion rate by around 17%.

***Index Terms*—Machine Learning Workflow, Large Language Model, Cloud**

I. INTRODUCTION

A workflow, commonly known as a data pipeline, entails a sequence of steps that process raw data from various sources, directing it to a destination for both storage and analysis. Similarly, an ML workflow streamlines the comprehensive MLOps workflow, spanning data acquisition, exploratory data analysis (EDA), data augmentation, model creation, and deployment. Post-deployment, this ML workflow facilitates reproducibility, tracking, and monitoring. Such workflows enhance the efficiency and management of the entire model lifecycle, leading to accelerated usability and streamlined deployment [31], [40]. To automate and oversee these workflows, ML orchestration tools are deployed, offering an intuitive and collaborative interface.

Mingjie Tang, Yuan Tang, and Qian Jiang performed most of this work while at Ant Group.

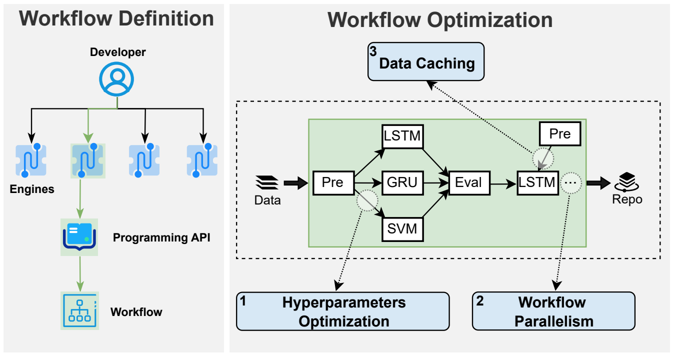


Fig. 1: An example of a financial company's journey in leveraging machine learning to predict market trends.

Example. Refer to the example in Figure 1, a financial company aims to predict market trends using ML models. Initially, developers are tasked with selecting a workflow engine from a variety of available options, such as Argo, Airflow, Dolphin Scheduler, MetaFlow or Kubeflow Pipeline etc. Then, end users need to dedicate time to mastering the programming API of specific workflow engines. Upon defining the workflow, the first step entails data preprocessing. Subsequently, three models are evaluated, and the most promising model, LSTM, is selected for further analysis. The preprocessed data is reloaded for subsequent analysis, culminating in the generation of a predictive report through a complex process. To implement this, the following challenges must be addressed:

- *How can a workflow description be automatically translated to an ML workflow?* For execution, developers must code the workflow to be compatible with different workflow engines. However, the guidelines for different workflow engines can vary significantly, posing a challenge to become proficient in all of them.
- *How can the built workflow be effectively optimized?* Given a well-defined workflow, optimization is crucial. Developers need to find the optimal hyperparameters for training the ML models, and manage workflow parallelism manually. In the absence of caching, both the data loader and intermediate results become critical points, potentially slowing down the process.

Goals and challenges. Given the ML workflow description and available resources, our objective is to autonomously construct a workflow that reduces dependence on expert knowledge. Simultaneously, we aim to enhance overall efficiency by minimizing end-to-end workflow execution costs. We strive to streamline the ML workflow creation process and ensure optimal utilization of available resources, making the entire system more user-friendly and efficient.

Effectively orchestrating workflows is crucial for companies heavily invested in machine learning. Consequently, a developer needs to understand the programming API of the selected workflow engine and learn to automate and optimize the entire workflow manually. Numerous widely used workflow engines exist, such as Argo Workflows [4], Tekton Pipelines [10], and Apache Airflow [1]. The necessity to master multiple workflow engines presents a significant challenge for developers due to the unique programming interface of each engine. With the advent of LLMs, significant strides has been made in the realms of natural language to SQL conversion [19], [36], [41] and code generation from natural language descriptions [33], [47]. This advancement facilitates the efficient conversion of natural language descriptions into programming coding across different workflow engines, thereby simplifying the workflow definition process. However, several challenges remain:

Given the myriad of available workflow engines, attempting a direct translation from NL to various workflow engine codes proves to be intricate and inefficient. Factors such as the continual evolution of workflow engine APIs and the distinct design philosophy behind each engine contribute to this complexity. Additionally, LLMs may not always stay updated with the latest changes in these APIs, posing a challenge to ensure accurate NL to code translation consistently. This scenario accentuates the need for a unified coding interface catering to different workflow engines. Such an interface simplifies the process of defining and managing workflows without delving into the intricacies of each engine, thereby enhancing the efficiency of LLMs in translating NL descriptions into executable code.

After establishing a workflow, optimizing its computational aspects is crucial. One challenge is to effectively cache intermediate results dynamically, maximizing resource use and minimizing runtime. Storing crucial intermediary outputs allows workflows to gracefully handle runtime errors without the need to restart from scratch. Moreover, splitting large workflows into smaller, more manageable segments is not straightforward. It demands careful strategizing to strike a balance between performance and resource use. In ML workflows, hyperparameter optimization of the models introduces another layer of complexity. Identifying the optimal hyperparameter values is a complex process, and leveraging the capabilities of LLMs to automate this process, while promising, remains a significant challenge.

Contributions. To address these challenges, our goal is to design a system for unified ML workflow optimization in the cloud. The contributions of this work are outlined below:

- **Simplicity and Extensibility**: We provide a unified programming interface for workflow definition, ensuring independence from the workflow engine and compatibility with various workflow engines such as Argo Workflows, Airflow, and Tekton. We demonstrate how COULER supports ML model selection and AutoML pipelines.
- **Automation**: We integrate LLMs in unified programming code generation. By leveraging LLMs, we facilitated the generation of unified programming code using NL descriptions. Additionally, we automate hyperparameters tuning through the integration of Dataset Card and Model Card, enhancing the effectiveness of the autoML process.
- **Efficiency**: We introduce the Intermediate Representative (IR) to depict the workflow Directed Acyclic Graph (DAG), optimizing extensive workflow computations by dividing a large workflow into smaller ones for auto-parallelism optimization. We also implement dynamic caching of artifacts, which are the outputs of jobs in the workflow, to minimize redundant computations and ensure fault tolerance.
- **Open Source Community**: We constructed the platform to assist data scientists in defining and managing workflows, enabling system deployment in real production environments on a large scale. The released open-source version has garnered adoption from multiple companies and end-users*. For instance, over 3000 end users are utilizing COULER within ANT GROUP , and more than 20 companies have adopted COULER as their default workflow engine interface.

II. SYSTEM FRAMEWORK

Figure 2 illustrates the COULER architecture, highlighting various components and multiple aggregation layers that facilitate scaling across clusters. Initially, we provide two interfaces for defining workflows: one through Natural Language and the other through GUI, SQL, and programming languages such as Python and GoLang. Once a workflow is defined, it's converted into an Intermediate Representation (IR) format. Subsequently, optimization measures, specifically the auto hyperparameter tuning optimizer and workflow auto-parallelism, are employed to refine the workflow. Upon completion, COULER generates the final workflow which is then submitted to the designated workflow engine. Concurrently, an automated caching mechanism operates in real-time, dynamically updating the cache as the workflow progresses.

A. Workflow Description

We offer two primary methods for users to construct workflows. The first leverages Natural Language (NL) descriptions, wherein we employ LLMs, such as ChatGPT-3.5 and ChatGPT-4, to generate code compliant with a standardized workflow interface definition (§III). Simultaneously, users can alternatively create workflows using a Graphical User Interface (GUI)(§V), SQL tools like SQLFlow(§V), or directly through programming languages such as Golang or Python.

*https://couler-proj.github.io/couler/

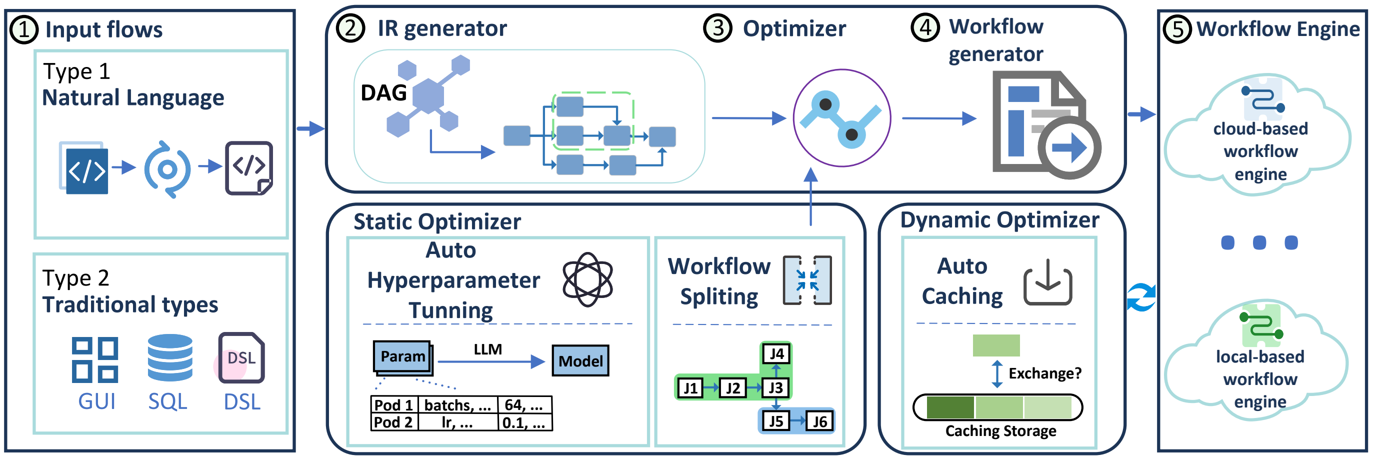


Fig. 2: Overview of COULER architecture.

B. *Workflow DAG Generator*

We propose a unified programming interface to define workflows in a DAG way. This interface is designed to allow users to delineate workflows without specific knowledge of the underlying workflow engine. And it offers fundamental functions such as executing scripts, containers, or jobs, stipulating conditions, and managing multiple instances of a job, among others. For example, the code 4 shows how to build a workflow implicitly. By this way, users need to own a clear big picture for the workflow, and under how the running logic among steps in their real application. The definition of DAG workflow via explicit way helps data engineer to debug a failed workflow more easily, and build a complicated workflow with hundred nodes. More detailed information about the interface is given in Section I of [9].

```python
def job(name):
    couler.run_container(
        image="whalesay:latest",
        command=["cowsay"],
        args=[name], step_name=name,)
def diamond():
    couler.dag(
    [[lambda: job(name="A")],
        [lambda: job(name="A"), # A -> B
           lambda: job(name="B")],
        [lambda: job(name="A"), # A -> C
           lambda: job(name="C")],
        [lambda: job(name="B"), # B -> D
           lambda: job(name="D")],
        [lambda: job(name="C"), # C -> D])
           lambda: job(name="D")]
    ])
```

Code 1: An Example Workflow DAG in COULER

C. *Workflow Intermediate Representation*

A workflow processes a stream of input data to train a model, subsequently generating a new model for machine learning applications. Typically, a workflow is represented in a DAG format. Consequently, we represent a workflow in an intermediate representation (IR) format, unbound to any specific backend workflow engine or platform. Utilizing IR allows us to optimize the workflow independently of platform-related properties, enabling COULER to assimilate workflows from the unified programming interface.

D. *Auto Tuning Optimizer and Workflow Optimizer*

We utilize LLMs to generate recommended hyperparameter configurations for machine learning models, by analyzing dataset characteristics from Dataset Card and model information from Model Card (§IV.C). This approach automates the fine-tuning of hyperparameters in machine learning workflows, enabling LLMs to generate configurations that enhance model performance.

Based on the workflow's IR, the COULER server employs a rule-based approach to formulate the optimization plan before initiating a workflow. The considerations for this plan include optimizing large workflows, resource request optimization, and the reuse of intermediate results. All optimizations adhere to a predefined interface, incorporating their specific implementations. Further details regarding these optimizations are provided in Section (§IV.B).

E. *Automatic Caching Optimizer*

In COULER , artifacts are integrated as valuable products of workflow development, including datasets, parameters, diagrams, etc. Various physical storage options are available and can be registered to accommodate different types of artifacts. We offer an Automatic Caching Mechanism based on the artifact to dynamically update the cache during workflow execution (§IV.A). For each currently executing pod, a comprehensive analysis is conducted across three dimensions: past usage, future usage, and the cost-effectiveness of caching. This analysis yields a cache value score, used to re-evaluate the existing cache content. This re-evaluation helps determine whether updates need to be made to the cache.

F. *Workflow Generator and Workflow Engines*

COULER aims to enable workflows to operate across various platforms, with a particular focus on cloud-native processing. To accelerate execution, we aim to support workflow generation tailored to specific platforms. As a result, the final phase of COULER optimization involves generating workflows to execute on distinct workflow engines. The workflow generator converts the intermediate representation of a DAG to an executable format. Then, a workflow engine like Argo can execute this format (e.g., YAML format for Argo workflow). This YAML is then sent to the Argo operator within a Kubernetes cluster, demonstrating how the abstraction of IR allows for flexibility in supporting various workflow engines.

In Kubernetes, the workflow engine operates as a workflow operator. Initially, this operator allocates the associated Kubernetes resources (i.e., Pods) according to the resource definition for a step in a workflow, and then monitors the status of steps, updating the workflow status as needed. The execution topology of the workflow is dictated by the workflow's DAG, with the workflow operator scheduling the relevant steps in the cluster based on the status of steps and the DAG.

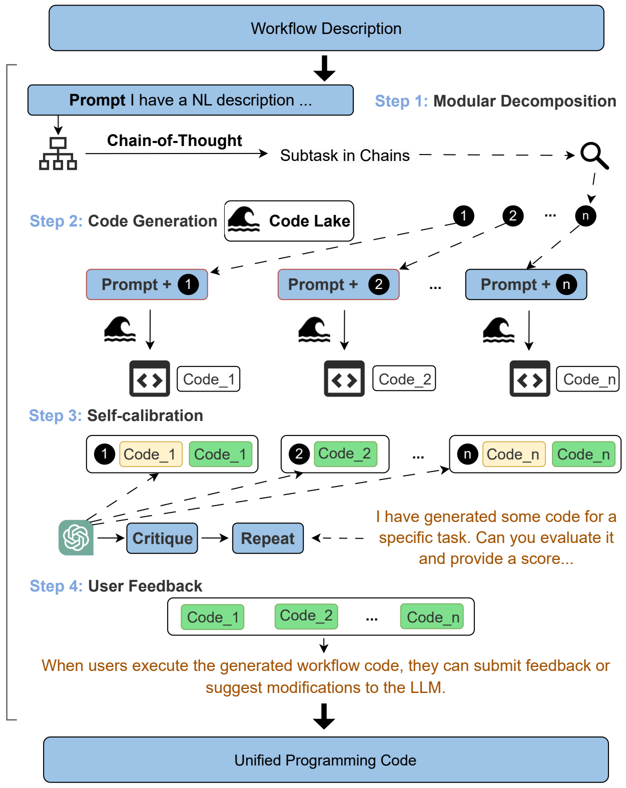
III. NL to Unified Programming Interface

In this section, we explore the application of LLMs for converting Natural Language (NL) to Unified Programming Interface as shown in Section(§III.B). Traditional methods involve defining workflows using various techniques and submitting them to a cluster. Lately, LLMs have demonstrated remarkable performance across a wide array of inference tasks. However, upon direct application of LLMs for unified programming code generation, certain challenges arise: Firstly, the overall workflow complexity hampers the performance of LLMs in complete workflow conversion. Secondly, LLMs possess limited knowledge regarding COULER 's unified programming interface.

To address these challenges, we introduce a method that leverages LLMs to automatically translate natural language into unified programming code via the crafting of task-specific prompts. This approach enables users to articulate their desired workflows in natural language, which are then automatically translated into executable unified programming code. As a result, our method simplifies the COULER workflow creation process and improves usability for individuals with limited programming experience, as illustrated in Figure 3. We also introduce this procedure through a running example in Section (§V.D). The transition from NL descriptions to COULER code encompasses four pivotal steps:

Step 1: Modular Decomposition: Initially, we employ a chain of thought strategy [46] to decompose natural language descriptions into smaller, more concise task modules, such as data loading, data processing, model generation, and evaluation metrics. Each module should encapsulate a singular, coherent task to ensure the precision and correctness of the generated COULER code. A series of predefined task types can be established to identify and extract pertinent tasks based on the input of natural language descriptions.

Step 2: Code Generation: For each independent subtask, we utilize LLMs to generate code. Considering that LLMs have limited knowledge about COULER , we construct a Code Lake containing code for various functions. We search for relevant code from the Code Lake for each subtask and provide it to



Workflow Description
Prompt I have a NL description ...
Step 1: Modular Decomposition
Chain-of-Thought
Subtask in Chains
Step 2: Code Generation
Code Lake
Prompt + 1
Prompt + 2
Prompt + n
Code_1
Code_2
Code_n
Step 3: Self-calibration
Critique
Repeat
I have generated some code for a specific task. Can you evaluate it and provide a score...
Step 4: User Feedback
When users execute the generated workflow code, they can submit feedback or suggest modifications to the LLM.
Unified Programming Code

Fig. 3: NL to Unified Programming Interface

LLMs for reference. This significantly improves the ability for unified programming code generation.

Step 3: self-calibration: After generating the code for each subtask, we integrate a self-calibration strategy [43] to optimize the generated code. This strategy evaluates the generated code by having LLMs critique it, as shown in Algorithm 1. Initially, we define a baseline score S_b as the standard evalua-

Algorithm 1 NL to Unified Programming Interface

```
Input: Description $\mathcal{D}$, LLM $\mathcal{L}$, Baseline_Score $S_b$
Output: Executable COULER Code $\mathcal{C}$
 1: Modular Decomposition: chain of thought to decompose NL description $\mathcal{D}$ into smaller chains $d_i$
 2: for each subtask $d_i$ in chains do
 3:     Generate Subtask Code:
 4:         Search for relevant code as reference
 5:         Use $\mathcal{L}$ to generate code $c_i$ for the subtask $d_i$
 6:     Self-calibration:
 7:         Compute score $s_i$ for $c_i$ leveraging $\mathcal{L}$
 8:     while $s_i < S_b$ do
 9:         Re-generate subtask code and update $s_i$
10:     end while
11: end for
12: User Feedback: review and validate the generated unified programming code
```

tion score. We use LLMs to evaluate the generated code c_i for a score s_i between 0 and 1, and if $s_i < S_b$, we will provide

feedback of LLMs and repeat the code generation. After this self-calibration, we will have improved code for each subtask.
Step 4: User Feedback: Finally, users can review and validate the generated workflow code. If the generated code fails to meet the users' requirements, they have the opportunity to provide feedback and suggestions. The system will leverage this feedback to optimize the code and enhance the precision of code generation.

IV. Workflow Optimization

In this section, we present three optimizations implemented at Ant Group to enhance workflow efficiency. Firstly, we introduce an artifact auto-caching mechanism to eliminate redundant computations. Secondly, for workflows comprising thousands of nodes, we propose a heuristic approach to partition large workflows into smaller units, thereby maximizing workflow parallelism. Lastly, we introduce an automatic hyperparameters tuning method based on LLMs to automate the training pipeline of ML workflows.

A. Automatic Artifact Caching Mechanisms

Motivation of Caching. Machine learning model development is a highly iterative process, often involving repeated steps with variations. This iterative nature can lead to significant duplicated work, especially concerning data import and transformation. By caching intermediate results, such as preprocessed data or feature representations, data scientists can avoid redundant computations across iterations, thus accelerating the development process. This increased iteration speed translates into higher productivity, allowing problems to be solved faster and empowering data scientists. However, caching introduces additional overhead e.g., storage costs. In this work, we introduce the way to strike the right balance between storage overhead and computational cost savings in Ant Group .

TABLE I: Set of common notations used in our description.

Notation	Definition
G	DAG of workflow ($G = \langle J, E, C\rangle$) Jobs J, Edges E, Configurations C
A	Adjacency matrix of a directed graph G
J_s, J_p	Serial and Parallel Job Sets
u	Artifact u
$\mathcal{L}$	The reconstruction cost of artifacts
$\mathcal{F}$	The utility value of artifacts
$\mathcal{V}$	The cache cost of artifacts
$\mathcal{I}$	The cache assessment metrics of artifacts
N_c	List of cached artifacts: $\{u_1, ..., u_i\}$
t, s	Computation Time and space usage of Jobs

1) Problem Statement and Evaluation Metrics: Caching all intermediate data (called artifacts in this work) is an instinctive approach, but it comes with challenges. Firstly, not all data merits caching, especially if it is not slated for reuse in the foreseeable future. Secondly, the associated costs of caching can be prohibitive. For instance, at Ant Group , we delegate intermediate artifact storage to distributed in-memory systems like Apache Alluxio [28]. Given the finite memory capacities of such systems, making judicious decisions about which artifacts to cache is crucial. This necessitates an automatic selection mechanism that factors in the caching expense cost when determining which data to store.

Thus, we prioritize workflow execution time and memory consumption as the pivotal performance metrics and targets for optimization. Specifically, we define the workflow execution time, represented as $\mathcal{T}$, as the duration required for completing the Critical Path. This Critical Path is characterized as the elongated sequence of interdependent tasks spanning from the inception to the culmination of the workflow as in [48]. On the other hand, the metric for memory expenditure, symbolized as $\mathcal{S}$, is construed as the peak memory consumption observed across all concurrently operating nodes. Based on these definitions, the cost function can be articulated as follows:

$$\mathcal{T} = \max(\sum_{p \in J_t} t_p) \tag{1}$$

$$\mathcal{S} = \max(\sum_{p \in J_s} s_p) \tag{2}$$

where t_p and s_p is the time and memory usage for Job p. We define the job groups with the longest running time and the largest resource consumption as J_t and J_s, respectively.

2) Principles of Automatic Caching: In this study, we propose a metric called the *caching importance factor* to ascertain the significance of caching a specific artifact (namely u). This factor serves as a guiding principle to dynamically determine which artifact warrants caching. We represent this by a function, $\mathcal{I}(u)$, which computes the *caching importance factor* for artifact u. Our formulation of this metric is primarily influenced by three determinants: the cost of reconstructing the artifact, denoted as $\mathcal{L}$; the expected value of reusing the artifact, represented as $\mathcal{F}$; and the associated expense of caching, labeled $\mathcal{V}$. Details are presented below.

Artifact reconstruction cost: refers to the expense incurred when re-creating or regenerating machine learning artifacts or intermediate results that were not cached or saved during the workflow. When these artifacts are not cached or saved, and they need to be reconstructed from raw data or recomputed, it can result in additional computational expenses, increased execution time, and potentially higher resource usage. Minimizing artifact reconstruction costs is one of the objectives of effective caching strategies in ML workflows.

In this research, given an artifact u, we focus on analyzing the subgraph containing nodes that serve as predecessors to artifact u, which we refer to as $G_p = \{J_1, ..., J_s\}$. Note that, to simply our discussion in this work, we only consider subgraphs with the following properties: (a) We select the subgraph G_p, formed by the preceding n layers of jobs from node u, as it is the most representative. (b)If the artifact of a job within G_p is cached, G_p will be truncated at that point. On this basis, we hope to minimize the related artifact reconstruction cost $\mathcal{L}(u)$. Within this subgraph G_p, the cost $\mathcal{L}(u)$ is determined by the computational resources utilized by jobs and the storage resources associated with the artifacts involved. Formally, $\mathcal{L}(u)$ is defined as follow way:

$$\mathcal{L}(u) = \sum_{i=1}^{s}\sum_{j=1}^{s} A_{ij} \cdot (w_i + d_i \cdot d_j) \tag{3}$$

where, A denote the adjacency matrix, respectively. w_i represents the resource consumption of job i. The degree d_i indicates the level of significance for job i, and s represents the number of nodes in G_p. By this way, we formula the overall runtime complexity of G_p , taking into account the varying importance of each node.

Artifact reuse value: refers to the benefits and advantages gained by reusing previously generated artifacts (e.g., preprocessed data, feature representations, or model checkpoints) in a machine learning workflow. The value comes from avoiding redundant computations and leveraging the work done in earlier stages of the workflow, ultimately leading to resource savings and more efficient model development. Maximum the reuse value is another optimization target in this work.

Given an artifact u, the artifact reuse value name as $\mathcal{F}(u)$ is influenced via the successor of workflow graph. This graph is referred as G_s whose definition is the same as G_p. Within this subgraph $G_s = \{J_1, ..., J_t\}$, we hope to maximize the artifact reuse value $\mathcal{F}(u)$ as following way.

$$\mathcal{F}(u) = \sum_{i=1}^{t} \frac{r}{\kappa_{ui}} \cdot (\zeta_{ui} + 1) \tag{4}$$

Where κ_{ui} represents distance for node u and node i in the subgraph G_s, r represents a boolean state indicating whether a reuse event occurs for artifact u and t represents the number of nodes in G_s. Then, ζ_{ui} is the weighted value for the dependency of job i on u. Given the adjacency matrix as A and the degree of nodes as d. We use $diag$ to represent the diagonal matrix, and matrix ζ can be computed as follow:

$$\zeta = diag[d_1, ..., d_n] - A \tag{5}$$

Artifact caching cost: refers to the expenses associated with storing and managing cached artifacts or intermediate results in a machine learning workflow. In this work, we use the distributed in-memory storage to store the artifact, thus, we mainly consider u's memory consumption (name as $\mathcal{V}(u)$).

Overall, given a artifact u, we formalize the *caching importance factor* of u as follow:

$$\mathcal{I}(u) = \alpha \cdot \log(1 + \mathcal{L}(u)) + \beta \cdot \mathcal{F}(u)^2 - e^{-\mathcal{V}(u)} \tag{6}$$

where α and β are weight parameters for the metrics, and their optimal values are selected through experimental studies in the production environment.

The *caching importance factor* plays a crucial role in deciding whether a new artifact should replace an existing one in the cache memory. This factor is instrumental in enabling COULER to maximize execution time efficiency while working within the constraints of limited cache space. In this way, we hope to reduce the communication overhead in the workflow and the reconstruction cost when artifacts are reused, thereby decreasing the overall runtime $\mathcal{T}$. We introduce the Algorithm 2 to determine which artifacts should be cached during the caching process based on the constraint.

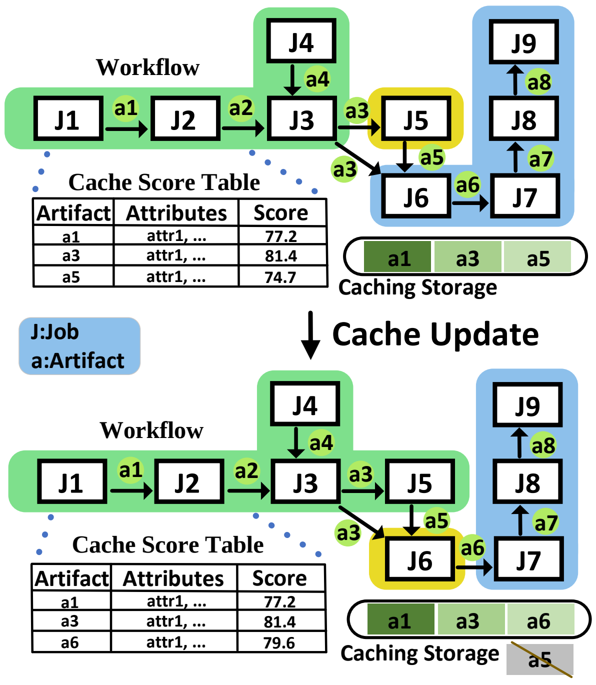
To make optimal cache exchange decisions, COULER 's dynamic caching module calculates the caching value of newly generated artifacts during the workflow execution process. Algorithm 2 provides an overview of how the dynamic caching strategy module makes cache decisions and optimizes execution time efficiency. The monitor attempts to place newly generated artifact into the cache (line 11). If there is insufficient cache space, we calculate a cache score based on the attributes of the new artifact (line 16-21). This score is then compared to the scores of artifacts already in the cache (line 24-30), determining whether to remove an existing cached artifact. This process is repeated until there is enough cache storage available or the score of the new artifact is lower than the compared score.

Algorithm 2 Automatic Caching Mechanisms

```
1:  Input: JobSet $\mathcal{N}$, Workflow $\mathcal{G}$, Artifact Cached List $N_c$,
    Used Caching Storage $C_u$, Total Caching Storage $C_t$
2:  Output: Dynamic Caching Set $D_c$
3:  function $\mathcal{L}(u)$ $\rightarrow$ Returns artifact reconstruction cost of $u$
4:  function $\mathcal{F}(u)$ $\rightarrow$ Returns artifact reuse value of $u$
5:  function $\mathcal{V}(u)$ $\rightarrow$ Returns artifact caching cost of $u$
6:  function $\mathcal{I}(l, f, v)$ $\rightarrow$ Returns caching importance factor
    of $u$
7:  $C_u \leftarrow \emptyset$, $N_c \leftarrow \emptyset$
8:  markUnVisited($\mathcal{G}$)
9:  for all $u \in \mathcal{N}$ do
10:     if not Visited($u$) and $C_u < C_t$ then
11:         $u \rightarrow N_c$
12:     else if not Visited($u$) and $C_u \geq C_t$ then
13:         NodeSelection($u$, $\mathcal{G}$, $N_c$, $C_u$, $C_t$)
14:     end if
15: end for
16: function NODESELECTION($u$, $\mathcal{G}$, $N_c$, $C_u$, $C_t$)
17:     for all $u \in N_c$ do
18:         $v_i \leftarrow \mathcal{V}(u)$            ▷ $\mathcal{V}(u)$:memory consumption
19:         $l_i \leftarrow \mathcal{L}(u)$                      ▷ using Eq. (3)
20:         $f_i \leftarrow \mathcal{F}(u)$                      ▷ using Eq. (4)
21:         $I_i \leftarrow \mathcal{I}(l_i, f_i, v_i)$              ▷ using Eq. (6)
22:     end for
23:     MarkVisited($u$)
24:     while $C_u > C_t$ do
25:         $u_{min} \leftarrow \arg\min_{u_i \in N_c} I_i$
26:         if $u_{min} \neq u$ then
27:             $u$ in $N_c$, $u_{min}$ out $N_c$
28:         else
29:             $u_i$ out $N_c$
30:         end if
31:         update $C_u$
32:     end while
33: end function
```

3) Running Example of Automatic Caching: Figure 4 presents a running example for the caching strategy in this work. In the workflow, the green sections represent the Jobs that have completed execution, the yellow sections indicate the Jobs currently in execution, and the blue sections denote the Jobs awaiting execution. Arrows represent the dependency relationships between Jobs. The Cache Score Table maintains records of the size, type, and other attributes of cached artifacts, as well as their cache scores. Note that caching storage refers to the cache space allocated for the workflow.

Upon the completion of $J6$, COULER calculates the cache score for artifact $a6$ based on the attribution of $a6$. Subsequently, COULER attempts to store artifact $a6$ in the Caching Storage. If the remaining space is to be insufficient for $a6$, COULER compares $a6$'s cache score with that of $a5$, which has the lowest score in the Cache Score Table. Due to $a6$'s score is higher than $a5$, $a5$ is replaced by $a6$. If the Caching Storage is still inadequate, the comparison continues with the artifact having the next lowest score, and this process is repeated until an artifact with a higher score than $a6$ or adequate storage capacity becomes available to cache $a6$.



Artifact	Attributes	Score
a1	attr1, ...	77.2
a3	attr1, ...	81.4
a5	attr1, ...	74.7

Artifact	Attributes	Score
a1	attr1, ...	77.2
a3	attr1, ...	81.4
a6	attr1, ...	79.6

Fig. 4: Running Example of Automatic Caching

B. Big Workflow Auto Parallelism Optimization

In general, a workflow can be very big (i.e., more than one thousand nodes). At ANT GROUP , we run into the case where the workflow involves more than four hundred nodes. This would bring two issues. At first, each workflow is a Kubernetes CRD (Custom Resource Definition), the CRD is defined in YAML format and the size of CRD is limited to specific requirements. For example, the API server of Kubernetes would be overflowed by the large CRD (e.g., the size of YAML can not bigger than 2MB in practice). Secondly, the user cannot define the workflows properly to achieve maximum parallelism in a big DAG, therefore, the optimizer of COULER needs to analyze the dependence of workflow and split the workflow into multiple ones.

Algorithm 3 Big Workflow Auto Parallelism Mechanisms

```
Input: Budget $\mathcal{C}$, Workflow $\mathcal{G}$
Output: Multiple split workflows $W_s$
1: Cand $\leftarrow \emptyset$, $W_s \leftarrow \emptyset$
2: markUnVisited($\mathcal{G}$)
3: for all $n_i \in \mathcal{N}$ do
4:     if not Visited($v_i$) then
5:         NodeSelection($n_i$, $\mathcal{G}$, $N_c$, $C_u$, $C_t$)
6:     end if
7: end for
8: function SPLITWORKFLOW($v_1$, $\mathcal{C}$, $\mathcal{G}, W_s, Cand$)
9:     $b_i \leftarrow BudgetOnUnVisitedVertex(\mathcal{G})$
10:    if ($b_1 \leq \mathcal{C}$) then
11:        $W_s \leftarrow W_s + \mathcal{G}$
12:        return $W_s$
13:    end if
14:    MarkVisited($v_i$)
15:    $\overline{C} \leftarrow$ Cand + $v_1$, $b_2 \leftarrow$ BudgetOnGraph( $\overline{C}$)
16:    if $b_2 \geq \mathcal{C}$ then
17:        $W_s \leftarrow W_s +$ Cand, Cand $\leftarrow v_1$
18:    else
19:        Cand $\leftarrow \overline{C}$
20:    end if
21:    for all $v \in adj(v_1)$ do
22:        if not Visited($v$) then
23:            SplitWorkflow($v$, $\mathcal{C}$, $\mathcal{G}, W_s$, Cand)
24:        end if
25:    end for
26: end function
```

In this paper, we first define the budget of workflow. The budget is used to decide whether we need to split a big workflow into small ones. The budget (namely $\mathcal{C}$) could be the (a) size of workflow CRD in YAML format: (α), (b) the number of steps in a workflow: β, (c) the number of pods: (γ) in a workflow. Thus, $\mathcal{C} = \alpha + \beta + \gamma$. In this work, we mainly use the size of workflow α as the default budget value. Naturally, if a workflow is bigger than a predefined budget, it needs to be split into small ones, otherwise, the optimizer would ignore this workflow.

Given the required budget and a big workflow in DAG format, the optimization goal is a problem of finding optimal DAG sets to schedule workflow so we can win the maximum parallel. It is tempting to reach for classical results [25] in the optimal graph topological order to identify an optimal schedule. The topological ordering of a directed graph could be used to split a big graph into smaller graphs for scheduling. In this work, we identify a workflow sets by deep first search (DFS) over a DAG. Algorithm 3 go through each vertex of the graph and put this vertex into a workflow candidates greedily until each vertex is visited or the workflow meeting the budget requirement. Initially, we mark every vertex as unvisited in line 1 and recursive split the related DAG from the unvisited vertex one by one from lines 2 to 4. Function $SplitWorkflow$ is used to split the input DAG. At first, we check whether the current workflow meets the requirement, that is, the budget is smaller than the requirement from lines 7 to 9. Next, we mark the current vertex v_1 as visited and check whether it is possible to add the vertex v_1 into the DAG candidate $\overline{C}$. if the vertex v_1 fails to join the current subgraph $\overline{C}$, we put the current subgraph $\overline{C}$ into the output set of DAGs (namely W_s). Finally, we go through the adjacent list of v_1 and continue to split the input DAG. Function $BudgetOnUnVisitedVertex$ in line 7 and $BudgetOnGraph$ in line 10 compute the related budget for the input graph for the un-visited vertex of input DAG or the whole DAG, respectively. Because we go through the input DAG via the depth first search order, the runtime cost of the proposed approach is the number of vertex (i.e., $O(|V|)$).

C. Automatic Hyperparameters Tuning

We explore the use of LLMs for automatic hyperparameters tuning of machine learning models by analyzing dataset characteristics from Dataset Card [20] and model information from Model Card [30]. This approach automates the fine-tuning of hyperparameters in machine learning workflows, enabling LLMs to generate configurations that enhance model performance. We detail the implementation approach and demonstrate how this automated configuration process improves the efficiency and effectiveness of model training in Algorithm 4.

Algorithm 4 Automatic Hyperparameters Tuning

```
Input: Data Card $\mathcal{D}$, Model Card $\mathcal{M}$, Hyperparameters Set $\mathcal{H}$, LLM $\mathcal{L}$
Output: Targeted Hyperparameters $h_t$
1: Data Card: comprise of the dataset name, input dataset type, label space, and default evaluation metrics
2: Model Card: consist of the model name, model structure, model descriptions, and architecture hyperparameters
3: for each hyperparameters $h_i$ in $\mathcal{H}$ do
4:     Predicted Training Log:
5:     /* Generate a training log $t_i$ for a given hyperparameter setting $h_i$ by leveraging LLM $\mathcal{L}$. */
6: end for
7: Targeted Hyperparameters:
8:     $h_t \leftarrow$ best performance for $h_i$ in $\mathcal{H}$ based on $t_i$
```

To fully exploit the capabilities of LLMs and generate effective prompts, we tailor prompts to the Data Card, Model Card, and hyperparameters information. The Data Card $\mathcal{D}$ comprehensively describes the dataset, including details such as data name, data type, label space, and evaluation metrics. The Model Card $\mathcal{M}$ provides a thorough description of the model, encompassing the model name, structure, description, and architecture hyperparameters, while the hyperparameters cover various parameter value ranges.

Initially, we have a hyperparameters set $\mathcal{H}$ containing a few optional hyperparameters. Without requiring training on actual hardware, we employ LLMs to automatically predict performance during the training process, subsequently returning a training log for each hyperparameters h_i in $\mathcal{H}$ [50]. This log captures various parameters and information from the training process. By examining the training log, we can observe the effects of different hyperparameters during training. After several rounds of testing, we select the training hyperparameters that yield the best performance. We introduce this procedure through a running example in [9].

V. Implementation

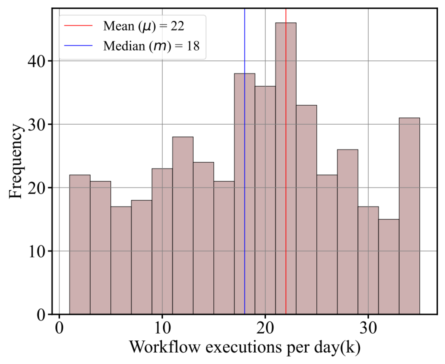
The Python SDK for Couler is now open-source. Several top enterprises have integrated this SDK into their production environments. The whole Couler service is crafted in Golang, encompassing all internal components. Initially, the service might remain proprietary due to user onboarding challenges, but enhancing and ensuring the reusability of the optimization components is our priority. Thus, we are developing these components as core libraries. Consequently, the service operates as a gRPC service atop these libraries. In its open-source form, the Python SDK can utilize these libraries for extended capabilities. It's worth mentioning that Couler is extensively used by Ant Group, managing over 20,000 workflows and 250,000 pods daily. Insights from our deployment experiences with the workflow engine are discussed in subsequent sections.

Monitor and failure handler. In order to reduce the unnecessary failure of workflow belonging to the system environment (i.e., abnormal patterns of cloud), we adopt following polices to improve the stability: (a) workflow on-time monitor, (b) workflow controller auto retry, (c) provide options for users to restart from failure. The details is shown in Section II.B of [9].

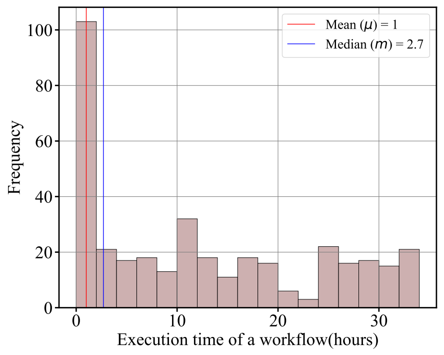
Caching input data for machine learning workflow. An analysis of production machine learning workflows at Ant Group, which include more than 5k models and 10k workflows for applications such as ads recommendation, fault detection, and video and image analysis, revealed that most workflows read the same data multiple times. We propose a new Kubernetes CRD, called *Dataset*, to represent the input and output data of a job. The details is shown in Section II.C of [9].

Interactive GUI. In addition to the programming API for defining workflows, we also offer a GUI interface within a web portal. With this approach, users can create workflows without any programming experience. The details is shown in Section II.D of [9].

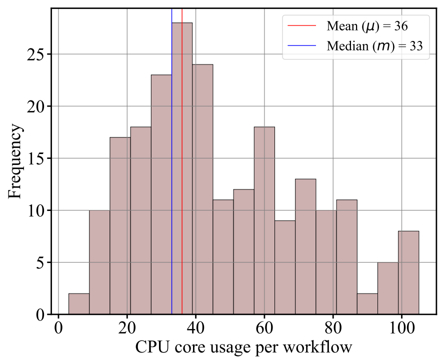
SQL and SQLFlow In addition to the GUI and Python programming interface, SQLFlow [45] offers an SQL-like language to train machine learning models and employ the trained models for predictions. The details is shown in Section II.E of [9].



(a) Distributin of workflow quantity

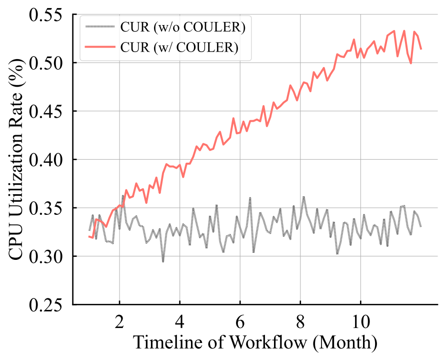


(b) Distributin of workflow lifespan

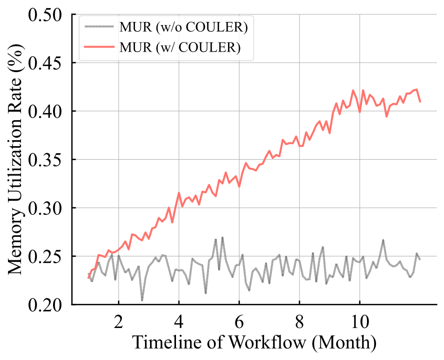


(c) Distributin of CPU core usage

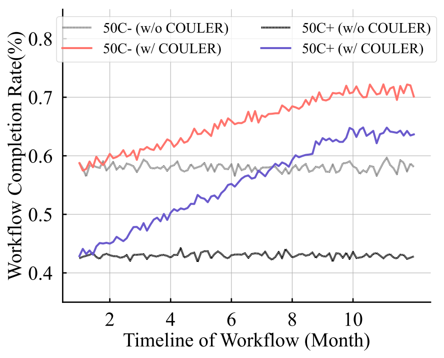
Fig. 5: From July 2022 to July 2023, workflow activity analysis of COULER in ANT GROUP



(a) Yearlong evolution of CUR



(b) Yearlong evolution of MUR



(c) Yearlong evolution of WCR

Fig. 6: From July 2022 to July 2023, 90% workflows in the cluster were transitioned to utilize COULER in ANT GROUP

VI. EVALUATION

Our evaluation results, which encompass diverse industrial models and data spanning several months, aim to address the following research questions:

- **RQ1:** What is the usage frequency of COULER in ANT GROUP ?
- **RQ2:** How effective is the automatic caching performance of COULER ?
- **RQ3:** How about the performance of NL to Unified Programming Code Generation?
- **RQ4:** How proficient is COULER 's capability of automatic hyperparameter configuration?

A. Experiment Setup

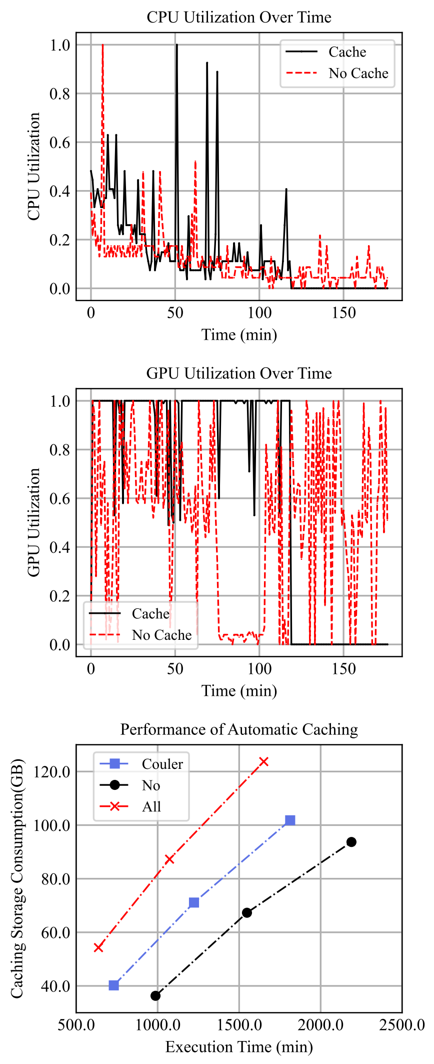
Production Environment. In ANT GROUP , various types of workflows operate concurrently in a shared cluster. The cluster provides substantial resources, with about 1,600,000 CPU cores, 4,500 GPU cores, 3.24 PB of memory, and 344 PB of disk space. This setup supports ANT GROUP 's diverse computational needs, enabling different workflows to run efficiently within the same shared resource environment. COULER is utilized to support over 95% of workflows in the production environment (e.g., 22k/day). The extensive scale of workflow operations provides accurate statistical estimates of the actual gain.

Workload. We design a multi-modal workflow in an isolated production environment to minimize interference of the production environment and conduct a more comprehensive assessment of COULER 's capabilities. By selecting appropriate component containers for model training, we evaluated the system's caching efficiency as well as the performance of its AutoML features.

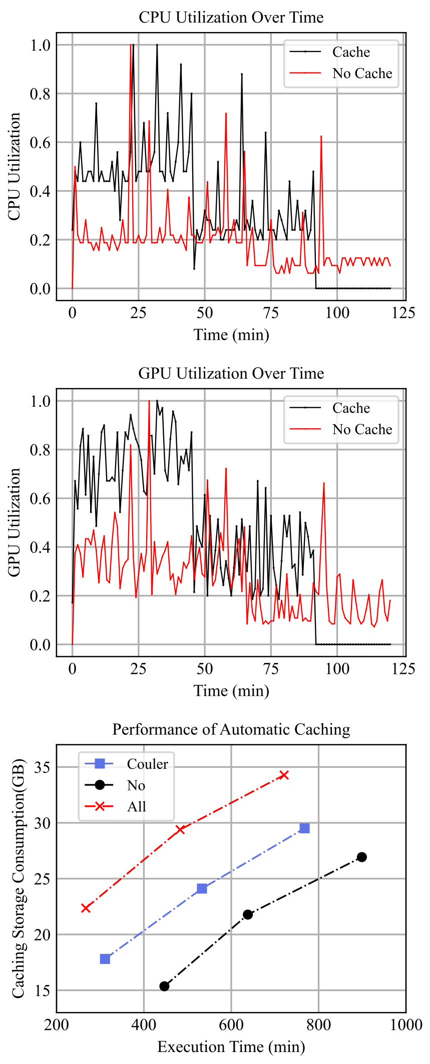
This workflow comprises two distinct tasks: image classification and language model fine-tuning. We tested the performance of models such as ViT and nanoGPT. Additionally, the workflow incorporates system testing modules and model update modules, increasing the task complexity to better emulate real-world scenarios. The workflow includes 26 different training scenarios and comprises 52 working pods, utilizing over 1.4 million images and 20GB of text data as datasets. Operating in contexts with a significant number of parameters and data volume, it effectively showcases COULER 's unique features.

B. Workflow Activity: RQ1

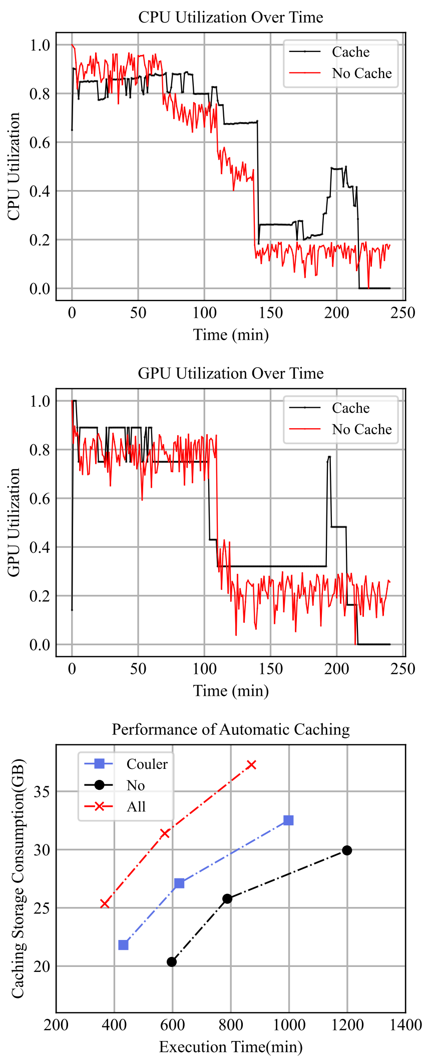
Initially, we explore three facets of ML workflows: daily usage frequency, typical lifespan, and CPU core usage. We focus on workflows within ANT GROUP from July 2022 to July 2023. Figure 5a illustrates the distribution of the average daily workflow count, revealing a daily average of 22,000 workflows within ANT GROUP . We define a workflow's lifespan as the hour count between the timestamps of its newest and oldest nodes in its trace, serving as an indicator of its active duration. Figure 5b depicts that, on average, a workflow within ANT



(a) Multimodal Training



(b) Image Segmentation



(c) Language Model Fine-tuning

Fig. 7: Effect of COULER on Resource Utils and Workflow Execution Time

GROUP remains active for 1 hour. Figure 5c presents the average CPU cores utilized by a workflow during its active period, with a mean of 36 cores being used per workflow in ANT GROUP .

To assess the effectiveness of COULER in optimizing workflows, we examine the evolution information of COULER within ANT GROUP from July 2022 to July 2023, as depicted in Figure 6. It took approximately ten months to execute all workflows with COULER . Figure 6a reveals that the CPU utilization rate (CUR) in machine learning workflow improved by 18%. Figure 6b shows that the memory utilization rate (MUR) improved by 17%. COULER 's enhanced fault tolerance significantly improved the workflow completion rate (WCR) for workflows running on 50- and 50+ CPU cores. Due to the high utilization rate of COULER , the CUR, MUR, and WCR have seen notable improvements. Therefore, COULER has effectively optimized ML workflow performance within ANT GROUP .

C. *Performance study with Caching: RQ2*

1) Performance study with Automatic caching: We evaluate the impact of COULER 's automatic caching strategy on workflow execution efficiency by comparing execution time and resource utilization against other caching strategies across three different scenarios:

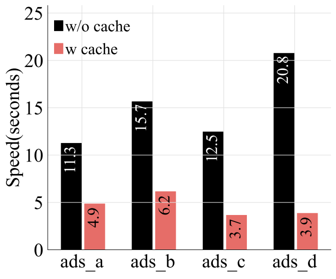
- **Multimodal Training**: This scenario encompasses 37 pods and 19 training models, and involves a training process that integrates various types of input data such as text, images, and sound, aimed at building more robust and adaptable models.
- **Image Segmentation**: This scenario includes 15 pods and 8 training models, focusing on segmenting digital images into multiple parts or sub-regions to identify and locate objects and boundaries within images.
- **Language Model Fine-tuning**: This scenario consists of 21 pods and 11 training models, primarily focusing on further training of pre-trained language models tailored

for specific tasks such as text classification or sentiment analysis.

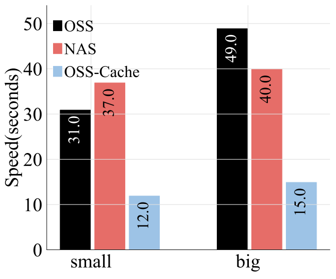
We evaluate the execution time and caching storage consumption across three different caching strategies as follows: (1) **No**, indicating no caching; (2) **ALL**, involving the caching of all data and intermediate results; (3) COULER , representing COULER 's automatic caching policy.

Figure 7 illustrates the variations in CPU and GPU usage over time, comparing COULER 's caching strategy with other caching strategies. It is evident that employing COULER 's caching strategy enhances GPU and CPU utilization, allowing the entire process to complete in less time. The scatter plot represents the overall execution time and resource consumption of workflows of varying sizes, indicating that COULER 's caching strategy achieves higher execution efficiency with a smaller additional resource cost.

2) Performance study with Data caching: We study how the cache improves the data reading performance. At first, we use the two tables (e.g., ads-a and ads-b) from the ads recommendation application used in internal, the data is partitioned and stored in the Alibaba ODPS([†]) with an approximate size of bigger than 10GB per partition. We show how the cache improves the data loading performance, as well as how the caching to improves the deep learning model training over CPU and GPU configuration. The cluster is a hybrid model with offline and online server computation in the same computation node. The reading Pod is configured with an 8 core CPU and 8 GB memory to test the data reading throughput, then the deep learning model is configured with 10 parameter servers and 20 workers. From Figure 8a, we can observe the cache can improve the data loading performance twice, this confirms the local storage can reduce the cost of remote network data accessing.



(a) Table cached



(b) File cached

Fig. 8: Performance of data caching

Similarly, we also study the performance of the caching for small and big file reading. Initially, these files are stored in the remote file system (Alibaba OSS([‡]) and NAS([§])). For the small files application, the number of files is more than 10k and the total size of files is more than 10GB. The size of a big file is more than 1GB with .zip format, and the total number of files is more than 10. We test the caching performance based on the different number of jobs. From Figure 8b, we observe the local cache would improve the data reading speed more than 4 times comparing the data without cache.

[†]https://www.alibabacloud.com/help/product/27797.htm
[‡]https://www.alibabacloud.com/product/oss
[§]https://www.alibabacloud.com/product/nas

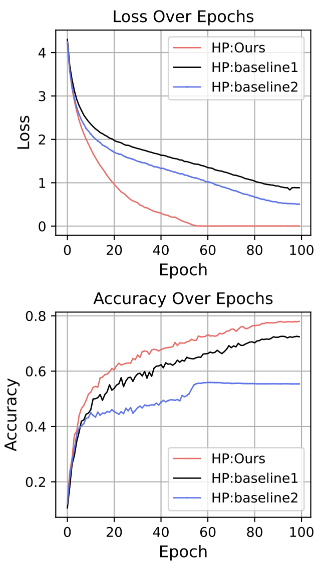
D. NL to Unified Programming Code Generation: RQ3

1) Experiment Result: We evaluate the effectiveness of utilizing LLMs to facilitate NL to Unified Programming Code Generation and compare our method with GPT-3.5 and GPT-4, as shown in Table II. All models are evaluated at temperatures $t \in \{0.2, 0.6, 0.8\}$, and we compute pass@k where $k \in \{1, 3, 5\}$ for each model. The temperature yielding the best-performing pass@k for each k is selected according to [33]. Our method significantly improves the performance of GPT-4 for NL to unified programming code generation and has been widely adopted for COULER code generation.

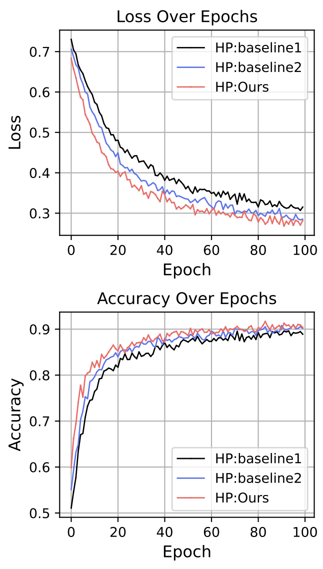
TABLE II: Evaluation results of our methods with GPT-3.5 and GPT-4. Each pass@k (where $k \in \{1, 3, 5\}$) for each model is computed with three sampling temperatures ($t \in \{0.2, 0.6, 0.8\}$) and the highest one among the three are displayed, which follows the evaluation procedure in [33].

Model	pass@k [%]		
	$k = 1$	$k = 3$	$k = 5$
GPT-3.5	35.21	37.19	39.21
GPT-4	45.81	48.11	50.23
GPT-3.5 + **Ours**	61.25	62.97	65.03
GPT-4 + **Ours**	**73.12**	**75.61**	**77.38**

2) Running Example: We illustrated an example, displaying the full automated process of converting Natural Language to COULER Code. The goal of this example is to choose the optimal image classification model from ResNet, ViT, and DenseNet, by showcasing the transformation from natural language descriptions to code generation. The details is shown in Section III of [9].



(a) Auto Configuration for CV



(b) Auto Configuration for NLP

Fig. 9: Effect of Auto Hyperparameter Configuration

E. Automatic Hyperparameter Configuration: RQ4

We next evaluate the performance of automatic hyperparameters configuration using LLMs to generate recommended hyperparameters. Following the workflow detailed in Workload, we apply automated hyperparameter tuning to both the *cv* and *nlp* modules, with *HP:Ours* as our recommended parameter. As depicted in Figure 9, the recommended parameters exhibit the lowest loss and the highest accuracy, showcasing the potent capability of COULER 's Automatic Hyperparameter Tuning.

VII. RELATED WORK

Jobs Scheduling in the Cloud. One machine learning workflow usually includes different stages to produce the model, and each stage/step is associated with different kinds of jobs [8], [13], [35]. Kubernetes [37] boasts a rapidly growing community and ecosystem, providing robust support for workflows. Kubernetes is based on a highly modular architecture that abstracts the underlying infrastructure and allows internal customization, such as the deployment of different software-defined network or storage solutions. It supports various big-data frameworks (e.g. Apache Hadoop MapReduce [7], [39], Apache Spark [48], Apache Kafka [2], Apache Flink [6] etc). More recently, Kubeflow [8] allows users to submit distributed machine learning tasks on Kubernetes such as TensorFlow, PyTorch, MPI, and XGBoost jobs a unified Kubernetes operator.

Workflow for AI/Machine Learning One machine learning workflow usually includes different stages to produce the model, and each stage/step is associated with different kinds of jobs [8], [13], [35]. Kubernetes [37] boasts a rapidly growing community and ecosystem, providing robust support for workflows. TFX [31] is a TensorFlow based AI framework for machine learning model training, but it is specifically designed for TensorFlow only. Some works in the HCI community study ML/DS workflows by interviewing ML developers and data scientists [49]. In this work, machine learning pipelines become more complex considering the hyper-parameter tuning and AutoML application [22]. In this work, we demonstrate how COULER can help end-users to do the hyper-parameter searching or AutoML like Katib [21].

Workflow Engine. A workflow engine is a software application that manages business processes. It is a key component in workflow technology and typically makes use of a database server. Argo Workflows [4] is an open-source container-native workflow engine for orchestrating parallel jobs on Kubernetes. Apache Airflow [1] is a Python-based platform for running directed acyclic graphs (DAGs) of tasks. Apache OOzie [3] is a workflow engine in the Hadoop ecosystem. Kubeflow [8] has a sub-project called Kubeflow Pipelines for end-users to develop machine learning pipelines. The complementary list of workflow engines can be found in link ¶. There are also some recently released workflow engines like Ray [32], CodeFlare [5] and ThunderML [38]. As more workflow engines come out, it is becoming much harder to choose a suitable workflow engine to use, not to mention that end users have to spend extensive efforts to get familiar with the related workflow engine to improve the performance of their workflows.

¶https://github.com/meirwah/awesome-workflow-engines

Automated Machine Learning. Automated Machine Learning (AutoML) [23] simplifies the process of machine learning model selection and hyper-parameter tuning, thereby making ML more accessible to non-experts. In the last decade, substantial advancements in AutoML have emerged with the introduction of open-source frameworks such as Auto-WEKA [27], [42], AutoSklearn [18], AutoGluon [17], and Auto-PyTorch [51], alongside commercialized frameworks. This growing body of work highlights the domain's potential to revolutionize ML practices. The increasing interest in the industrial applications of AutoML, attributable to its ability to conserve time and effort on repetitive tasks in ML pipelines, underscores its value in enhancing the efficiency of ML engineers across various industries [44].

Programming Model. For programming over cluster, data flow models such as MapReduce [16], and Dryad [24] support a rich set of operators for processing data but share it through stable storage systems. Pregel [29] and Spark [48] provide high-level interfaces for specific classes of applications requiring data sharing. Several systems like Spark SQL [12] and Flink [6] have sought to combine relational processing with the procedural processing engines initially used for large clusters. COULER is inspired by the design of PyTorch [34], which compiles a high-level AI model to a DAG and running over a distributed DAG execution engine. COULER goes beyond PyTorch by also providing a programming interface for scheduling different types of jobs in cloud.

VIII. CONCLUSION

In this paper, we introduced COULER , a system designed for unified machine learning workflow optimization in the cloud. COULER simplifies ML workflow generation using NL descriptions, abstracting the complexities associated with different workflow engines. The successful deployment of COULER has led to notable improvements in resource utilization, with over a 15% increase in CPU/Memory utilization and approximately a 17% improvement in workflow completion rates.

IX. ACKNOWLEDGEMENT

We want to thank Wei Yan who led the initial discussions of the design and implementation of this project. We would like to thank many colleagues at Alibaba Group and Ant Group for their valuable advice that improved this project, namely, Lan Li, Jian Liu, Wei Shu, Changhua He, Yi Wang, Jun Jiang, Chunbo Dai, Chenyi Ding, Mu Xiong, Yitao Shen, Yi Zhang, Xiaohua Cai, and Qinglong Wang. In addition, we appreciate the timely review and feedback from the Argo team every time when we contributed bug fixes and new features to not only meet our internal requirements but benefit the community users. Last but not least, we are grateful for the contributions from the COULER open source community and their initial public adoption and feedback that improves the project.

References

[1] "Airflow: a workflow management platform," https://airflow.apache.org/, Oct. 2023.

[2] "Apache kafka," https://kafka.apache.org/, Oct. 2023.

[3] "Apache ooize," https://oozie.apache.org/, Oct. 2023.

[4] "Argo workflows," https://argoproj.github.io/argo-workflows/, Oct. 2023.

[5] "Codeflare," https://codeflare.readthedocs.io/en/latest/getting_started/overview.html, Oct. 2023.

[6] "Flink," http://flink.apache.org/, Oct. 2023.

[7] "Hadoop," http://hadoop.apache.org/, Oct. 2023.

[8] "Kubeflow," https://www.kubeflow.org/, Oct. 2023.

[9] "Technical report of couler: Optimizing machine learning workflows in cloud," https://github.com/ignite-abd/Technical-Report-of-Couler, Oct. 2023.

[10] "Tekton," https://tekton.dev/, Oct. 2023.

[11] "Xgboost," https://github.com/dmlc/xgboost, Oct. 2023.

[12] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015, pp. 1383–1394.

[13] A. Chen, A. Chow, A. Davidson, A. DCunha, A. Ghodsi, S. A. Hong, A. Konwinski, C. Mewald, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, A. Singh, F. Xie, M. Zaharia, R. Zang, J. Zheng, and C. Zumar, "Developments in mlflow: A system to accelerate the machine learning lifecycle," ser. DEEM'20. New York, NY, USA: Association for Computing Machinery, 2020.

[14] H.-T. Cheng, Z. Haque, L. Hong, M. Ispir, C. Mewald, I. Polosukhin, G. Roumpos, D. Sculley, J. Smith, D. Soergel, Y. Tang *et al.*, "Tensorflow estimators: Managing simplicity vs. flexibility in high-level machine learning frameworks," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1763–1771.

[15] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & deep learning for recommender systems," 2016.

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[17] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[18] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," *Advances in neural information processing systems*, vol. 28, 2015.

[19] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-sql empowered by large language models: A benchmark evaluation," *arXiv preprint arXiv:2308.15363*, 2023.

[20] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. D. Iii, and K. Crawford, "Datasheets for datasets," *Communications of the ACM*, vol. 64, no. 12, pp. 86–92, 2021.

[21] J. George, C. Gao, R. Liu, H. G. Liu, Y. Tang, R. Pydipaty, and A. K. Saha, "A scalable and cloud-native hyperparameter tuning system," *arXiv preprint arXiv:2006.02085*, 2020.

[22] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018, in press, available at http://automl.org/book.

[23] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 5972.

[25] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, p. 558562, Nov. 1962.

[26] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Red Hook, NY, USA, 2017, p. 31493157.

[27] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-weka: Automatic model selection and hyperparameter optimization in weka," *Automated machine learning: methods, systems, challenges*, pp. 81–95, 2019.

[28] H. Li, *Alluxio: A virtual distributed file system*. University of California, Berkeley, 2018.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, New York, NY, USA, 2010, pp. 135–146.

[30] M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru, "Model cards for model reporting," in *Proceedings of the conference on fairness, accountability, and transparency*, 2019, pp. 220–229.

[31] A. N. Modi, C. Y. Koo, C. Y. Foo, C. Mewald, D. M. Baylor, E. Breck, H.-T. Cheng, J. Wilkiewicz, L. Koc, L. Lew, M. A. Zinkevich, M. Wicke, M. Ispir, N. Polyzotis, N. Fiedel, S. E. Haykal, S. Whang, S. Roy, S. Ramesh, V. Jain, X. Zhang, and Z. Haque, "Tfx: A tensorflow-based production-scale machine learning platform," in *KDD 2017*, 2017.

[32] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[33] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[35] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data lifecycle challenges in production machine learning: A survey," *SIGMOD Rec.*, vol. 47, no. 2, p. 1728, Dec. 2018.

[36] N. Rajkumar, R. Li, and D. Bahdanau, "Evaluating the text-to-sql capabilities of large language models," *arXiv preprint arXiv:2204.00498*, 2022.

[37] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[38] S. Shrivastava, D. Patel, W. M. Gifford, S. Siegel, and J. Kalagnanam, "Thunderml: A toolkit for enabling ai/ml models on cloud for industry 4.0," in *International Conference on Web Services*. Springer, 2019, pp. 163–180.

[39] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[40] E. Sparks, S. Venkataraman, T. Kaftan, M. Franklin, and B. Recht, "Keystoneml: Optimizing pipelines for large-scale advanced analytics," 04 2017, pp. 535–546.

[41] R. Sun, S. O. Arik, H. Nakhost, H. Dai, R. Sinha, P. Yin, and T. Pfister, "Sql-palm: Improved large language modeladaptation for text-to-sql," *arXiv preprint arXiv:2306.00739*, 2023.

[42] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 847–855.

[43] K. Tian, E. Mitchell, A. Zhou, A. Sharma, R. Rafailov, H. Yao, C. Finn, and C. D. Manning, "Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback," *arXiv preprint arXiv:2305.14975*, 2023.

[44] A. Truong, A. Walters, J. Goodsitt, K. Hines, C. B. Bruss, and R. Farivar, "Towards automated machine learning: Evaluation and comparison of automl approaches and tools," in *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)*. IEEE, 2019, pp. 1471–1479.

[45] Y. Wang, Y. Yang, W. Zhu, Y. Wu, X. Yan, Y. Liu, Y. Wang, L. Xie, Z. Gao, W. Zhu, X. Chen, W. Yan, M. Tang, and Y. Tang, "Sqlflow: A bridge between sql and machine learning," 2020.

[46] 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*, vol. 35, pp. 24 824–24 837, 2022.
[47] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.
[48] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28.
[49] A. X. Zhang, M. Muller, and D. Wang, “How do data science workers collaborate? roles, workflows, and tools,” 2020.
[50] S. Zhang, C. Gong, L. Wu, X. Liu, and M. Zhou, “Automl-gpt: Automatic machine learning with gpt,” *arXiv preprint arXiv:2305.02499*, 2023.
[51] L. Zimmer, M. Lindauer, and F. Hutter, “Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 9, pp. 3079–3090, 2021.

Appendix A
Unified programming model

This section provides an overview of the programming model to define a workflow. We at first introduce the programming interface, then show some examples for running a workflow in a machine learning application. The major design rule of COULER is aiming to help end-users to write a workflow without the specific knowledge of the workflow engine itself. In this paper, we provide two ways to define a workflow. One way is building a workflow implicitly (e.g., code 2 and code 3), the other way is explicitly defining a workflow as code 4. The main difference is whether DAG is described in the program. The core functions of COULER are listed in Table III, we would illustrate how to use COULER to build a workflow based on the following examples. Because most data scientists prefer to use Python, the programming interface of COULER is based on Python. However, this design is not limited to Python and could be extended to Java or another language. At Ant Group, we also provided Java client for end-users.

Name	API	Description
Run script	*couler.run_script*()	Run a script in a Pod
Run container	*couler.run_container*()	Start a container
Run job	*couler.run_job*()	Start a distributed job
Condition	*couler.when*()	Condition definition
Map	*couler.map*()	Start multiple instances for one job
Concurrent	*couler.concurrent*()	Run multiple jobs at the same time
Recursive	*couler.exec_while*()	Run a function until a condition meets

TABLE III: API Summary of COULER

```
1  def producer(step_name):
2     output_path = "/opt/hello_world.txt"
3     output_place =
4        couler.create_parameter_artifact(
5         path=output_path, is_global=True
6     )
7     return couler.run_container(
8        image="docker/whalesay:latest",
9        args=["echo -n hello world >
10       %s" % output_place.path],
11       command=["bash", "-c"],
12       output=output_place,
13       step_name=step_name,
14    )
15
16 def consumer(step_name, input):
17    couler.run_container(
18       image="docker/whalesay:latest",
19       command=["cowsay"],
20       step_name=step_name,
21    )
22
23 output = producer("step1")
24 consumer("step2", output)
```

Code 2: Basic workflow and artifact definition in COULER

Name	API	Description
Parameter	*couler.create_parameter_artifact*	Create a parameter
HDFS	*couler.create_hdfs_artifact*	Create a HDFS artifact
Amazon S3	*couler.create_s3_artifact*	Create a S3 artifact
Alibaba OSS	*couler.create_oss_artifact*	Create a OSS artifact
Google GCS	*couler.create_gcs_artifact*	Create a GCS artifact
Git storage	*couler.create_git_artifact*	Create a Git artifact

TABLE IV: Artifact support in COULER

A. *Basic workflow example and artifact*

A workflow is made by different steps, then each step is isolated from each other in the cloud base on the container. A container manages the complete the lifecycle of its host system, the contained environment helps each step to own the specific computing requirement and resource. However, this brings issues to pass data from one step to the following step in a workflow. In this work, we introduce the artifact to help users to store the intermediate results inside a workflow.

An artifact is a by-product of workflow development and created. This might include things like data set, parameter, and diagram, etc. For example, a machine learning pipeline generates statistic results, trained models, or new features. For different kinds of artifacts, users can register different physical storage to place the related artifact based on specific requirements. Thus, we introduce multiple ways to store artifacts as Table IV.

Take the code 2 as a running example, users register a parameter artifact to pass data among two steps (e.g., a producer and consumer). Function $producer()$ generate a message and pass the message to function $consumer()$ from line 1 to line 15. Each function is built based on $couler.run\_container()$, where $couler.run\_container()$ is used to start a Pod in Kubernetes and run the corresponding function. As a result, two Pods start and run step by step in the workflow as running a local python code lines 22 to 23. Then, the workflow engine propagates the related value among steps without users’ interaction. Note, Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

B. *Control flow*

```
1  def random_code():
2     import random
3
4     res = "heads" if random.randint(0, 1)
5        == 0
6     else "tails"
7     print(res)
8
9  def flip_coin():
10    return couler.run_script(
11       image="python:alpine3.6",
12       source=random_code)
13
14 def heads():
15    return couler.run_container(
16       image="alpine:3.6",
17       command=["sh", "-c",
18       'echo "it was headed"']
19    )
20
21 def tails():
22    return couler.run_container(
23       image="alpine:3.6",
24       command=["sh", "-c",
25       'echo "it was tailed"']
26    )
27
28 result = flip_coin()
29 couler.when(couler.equal(result, "heads"),
30   lambda: heads())
31 couler.when(couler.equal(result, "tails"),
32   lambda: tails())
```

Code 3: Workflow control in COULER

Control flow is important for defining a workflow. For a machine learning workflow, if a trained model fails to meet the online serving criteria, the following step (e.g., model deployment step) would not push the model to a server rather than alerting user the model training failures.

This example code 3 combines the use of a Python function result (e.g., Function *random_code*()), along with conditionals, to take a dynamic path in the workflow. In this example, depending on the result of the first step defined in *flip_coin* (line 27) , the following step will either run the *heads*() step (line 28) or the *tails*() step (line 29). We can notice, steps in COULER can be defined via either Python functions or script to running for containers (e.g., line 9). In addition, the conditional logic to decide whether to flip the coin in this example is defined via the combined use of *couler.when*() and *couler.equal*(). As a result, users can control the workflow logic based on the results of steps in the workflow dynamically.

C. *Define a workflow explicitly*

```
1  def job(name):
2     couler.run_container(
3        image="docker/whalesay:latest",
4        command=["cowsay"],
5        args=[name],
6        step_name=name,
7     )
8
9  #   A
10 #  / \
11 # B   C
12 #  \ /
13 #   D
14 def diamond():
15    couler.dag(
16       [
17          [lambda: job(name="A")],
18          [lambda: job(name="A"),
19             lambda: job(name="B")], # A ->
               B
20          [lambda: job(name="A"),
21             lambda: job(name="C")], # A ->
               C
22          [lambda: job(name="B"),
23             lambda: job(name="D")], # B ->
               D
24          [lambda: job(name="C"),
25             lambda: job(name="D")], # C ->
               D
26       ]
27    )
28 diamond()
```

Code 4: Workflow DAG in COULER

In general, data scientists prefer to build a workflow implicitly as the examples mentioned above. On other hand, data engineers want to organize a workflow explicitly. COULER support end-users to build the dependency among steps based on function *set_dependencies*. Function *set_dependencies* take input as a function and let the user to define the dependencies steps of others based on the step name. For example, the code 4 generates a diamond workflow as line 15. In this way, users need to own a clear big picture for the workflow, and under how the running logic among steps in their real application. At Ant Group, we analyze the users' preference to choose to build a workflow and we found major of data scientists choose to define a workflow implicitly, yet, data engineers incline to build a workflow explicitly since they facing more than one hundred steps in a workflow. The definition of DAG workflow via explicit way helps data engineer to debug a failed workflow more easily, and build a complicated workflow with hundred nodes.

D. *Example: running recursive in a workflow*

COULER provide a straightforward way to help end users to define the recursive logic in a workflow. For machine learning workflow, data scientists need to search a machine learning model until the model meets the specific requirement (e.g., model precision, convergence ratio, or the number of iteration steps is bigger than predefined value). Thus, data scientist hope to search the best ML model in a workflow recursively. Example 5 demonstrates how to run the recursive in a workflow. This *flip_coin*() step is running recursively until the output is equal a *tails* (line 14 to 15).

```
1  def random_code():
2     import random
3
```

```
    result = "heads" if random.randint(0, 1)
       == 0 else "tails"
    print(result)


def flip_coin():
    return couler.run_script(
       image="alpine3.6",
       source=random_code)

# Stop flipping coin until the outputs of
    'flip_coin' is not 'tails'
couler.exec_while(couler.equal("tails"),
    lambda: flip_coin())
```

Code 5: Recursive in COULER

E. Example: select a best ML model

```
1  def train_tensorflow(batch_size):
2     import couler.steps.tensorflow as tf
3
4     return tf.train(
5        num_ps=1,
6        num_workers=1,
7        command="python /train_model.py",
8        image="wide-deep-model:v1.0",
9        input_batch_size=batch_size,
10    )
11
12
13 def run_multiple_jobs(num_jobs):
14    para = []
15    i = 0
16    batch_size = 0
17    while i < num_jobs:
18       batch_size += 100
19       para.append(batch_size)
20       i = i + 1
21
22    return couler.map(lambda x:
23       train_tensorflow(x), para)
24
25 def evaluation(model_path):
26    return couler.run_container(
27       image="model_evalutation:v1",
28       command=["python model_eval.py"],
29       args=[model_path],
30       step_name="eval",
31    )
32
33 model_path = run_multiple_jobs(5)
34 couler.map(lambda x: evaluation(x),
35    model_path)
```

Code 6: Searching a best ML model in COULER

The hyper-parameters of machine learning modes such as batch size or converge ratio decide the performance of the model. Data scientists need to run multiple jobs in one same workflow to find the best model based on the same input data.

The sample program 6 implements a model searching procedure for a deep learning model (e.g., wide and deep model [15]. This is a common recommendation algorithm that recommends items to users based on users' profiles and user-item interaction. We start by defining a training job via the step zoo of COULER , this job train a DL model based on the different batch size from line 1 to 10, then run *map* function to start multiple TensorFlow jobs in the same workflow from line 13 to 33. Next, multiple evaluation steps are running based on the outputs of previous model training results from line 25 to 31.

F. Example: Running an AutoML pipeline

```
1  def train_xgboost():
2      train_data = Dataset(
3        table_name="pai_telco_demo_data",
4        feature_cols="tenure,age,
5           marital,address,ed,employ",
6        label_col="churn",
7     )
8
9     model_params = {"objective":
10       "binary:logistic"}
11    train_params = {"num_boost_round": 10,
12       "max_depth": 5}
13
14    return xgboost.train(
15       datasource=train_data,
16       model_params=model_params,
17       train_params=train_params,
18       image="xgboost-image",
19    )
20
21 def train_lgbm():
22    train_data = Dataset(
23       table_name="pai_telco_demo_data",
24       feature_cols="tenure,age,
25          marital,address,ed,employ",
26       label_col="churn",
27    )
28
29    lgb = LightGBMEstimator()
30    lgb.set_hyperparameters(num_leaves=63,
31       num_iterations=200)
32    lgb.model_path = "lightgbm_model"
33    return lgb.fit(train_data)
34
35 couler.concurrent([lambda: train_xgboost(),
36    lambda: train_lgbm()])
```

Code 7: An AutoML workflow in COULER

A more complex machine learning workflow application is the AutoML. Different from hyper-parameter tuning, data scientist prefers to select the best models from multiple model candidates based on the same input data. This program 7 shows how to choose a best model from two machine learning model (e.g., XGBoost [11] and LightGBM [26]), which are the state-of-art tree and boost based machine learning model. The training model in $train\_xgboost()$ and $train\_lightbm()$ is defined based on user's from line 1 to line 33, then $couler.concurrent()$ will run two jobs parallel in the same workflow. Different from the way of $couler.map()$, Function $couler.concurrent()$ start two training process based on different machine learning model.

Appendix B
Implementation

The Python SDK for COULER is now open-source, as shown in its public repository [‖]. Several top enterprises have integrated this SDK into their production environments. The whole COULER service is crafted in Golang, encompassing all internal components. Initially, the service might remain proprietary due to user onboarding challenges, but enhancing and ensuring the reusability of the optimization components is our priority. Thus, we are developing these components as core libraries. Consequently, the service operates as a gRPC service atop these libraries. In its open-source form, the Python SDK can utilize these libraries for extended capabilities.

It's worth mentioning that COULER is extensively used by Ant Group, managing over 20,000 workflows and 250,000 pods daily. Insights from our deployment experiences with the workflow engine are discussed in subsequent sections.

A. *Workflow scheduling among clusters*

to be confirmed At Ant Group, we have more than one clusters in different locations. Workflows are scheduled among those clusters and each cluster has its specific configuration. For example, Cluster A is specifically designed for GPU jobs, Cluster B is located far away from the storage cluster, Cluster C provides more CPU capacity than others. In addition, the storage and computation capacity of a cluster is changing with respect the time. We need to make sure each cluster owns the similar computation load. In this work, we provide a workflow queue to schedule the related steps of workflow into a corresponding cluster based on the following properties: (a) the workflow priority based on business logic, (b) cluster current capacity of CPU/Memory, (c) user's current CPU/Memory quota, (d) user's current GPU quota. Then, a job is queued and pull out from the queue based on the weight combination of mentioned factors. In this way, we can guarantee each cluster shares a similar capacity and avoid one cluster being overflow in the production.

B. *Monitor and failure handler*

In order to reduce the unnecessary failure of workflow belonging to the system environment (i.e., abnormal patterns of cloud), we adopt following polices to improve the stability: (a) workflow on-time monitor, (b) workflow controller auto retry, (c) provide options for users to restart from failure.

Initially, we monitor workflow status and track the health status of the workflow engine. For example, we record the number of workflows based on their status, the latency for the workflow operator to process a workflow, etc. This monitor metric helps the SRE to respond to the abnormal behaviors of the workflow at the first time.

Subsequently, we get the patterns of system errors related workflow. For example, "ExceededQuotaErr" means the Etcd of Kubernetes exceeded quota during the system is updating. "TooManyRequestsErr" means too many requests being handled by API-server, usually happens under high pressure. The backoff limit retry policy would help avoid DDOS of the cluster Etcd server. In general, we have found more than 20 abnormal patterns to retry, then the workflow controller restarts the failed step inside a workflow rather than from the begging automatically.

Furthermore, there are instances where users prefer to manually retry some failed workflows, a scenario frequently encountered in machine learning. In such cases, data scientists update the relevant steps and wish to retry the workflow from the failure point instead of from the beginning. To address this type of failure, COULER 's server first retrieves the failed workflow from the database. Note that we persist workflow metadata into a database for automated management. The server then processes the failed workflow, skipping the steps with "Succeeded," "Skipped," or "Cached" status. Subsequently, the server deletes the failed steps and the related CRDs and marks these steps as running. Finally, the workflow's status is updated to running, and it is restarted from the failed step by the workflow operator.

C. *Caching input data for machine learning workflow*

In a machine learning workflow, the input data for model training is stored in a data storage cluster, while the machine learning job runs in a separate computation cluster. This necessitates fetching data from remote storage before training, which is time-consuming and can lead to network IO failures. This is particularly problematic for applications like ads recommendation, where input tables often exceed 1TB, and for image and video deep learning models, which usually involve over a million files.

An analysis of production machine learning workflows at Ant Group, which include more than 5k models and 10k workflows for applications such as ads recommendation, fault detection, and video and image analysis, revealed that most workflows read the same data multiple times. For instance, 70% and 85% of the input data for tables and files, respectively, was read repeatedly. This redundancy arises due to several factors: (1) a single training job may need to scan the entire dataset multiple epochs, (2) multiple training jobs may need to read the same data to train the basic model, and (3) different training jobs may read overlapping data partitions using a sliding time window.

Currently, users read input data via Python/Java clients in the Kubernetes pods of machine learning training jobs. However, the workflow engine, such as Argo Workflows, cannot track data flow because it operates on Kubernetes Custom Resource Definitions (CRD) and does not monitor the runtime information of pods. This leads to two issues: (1) if a workflow fails, the training job must read the input data again, and (2) if multiple training jobs in the same workflow read the same input data, each job reads the data remotely, leading to redundant data access and high network IO.

To address these issues, we propose a new Kubernetes CRD, called *Dataset*, to represent the input and output data of a job. The schema of *Dataset* is shown in Code 8. This CRD

[‖]https://couler-proj.github.io/couler/

records the metadata of the data, enabling the workflow engine to understand the input and output of a training job and skip steps to read cached data. Additionally, a caching server reads the *Dataset* status and syncs the data from the storage cluster to the computation cluster, eliminating the need for multiple data synchronizations for different jobs.

```
apiVersion:
    io.kubemaker.alipay.com/v1alpha1
kind: Dataset
metadata:
  name: couler-cache-dataset
spec:
  owner: user_id
  odps:
    accessID:
      secretKeyRef:
        name: test-dataset-secret
        key: aid
    accessKey:
      secretKeyRef:
        name: test-dataset-secret
        key: akey
    project: test_project
    table: test_table
```

Code 8: Dataset CRD in COULER

D. Interactive GUI

In addition to the programming API for defining workflows, we also offer a GUI interface within a web portal. With this approach, users can create workflows without any programming experience. Let's consider Figure 10 as an example. Users aim to identify the best model for predicting user churn. Data scientists initially define data splitting methods for training, select various well-known models (e.g., logistic regression, random forest, and XGBoost) for training the same data, and ultimately choose the best model based on evaluation results. End-users only need to configure model-related parameters or data splitting methods. The backend then translates these actions into the workflow's IR, as explained in Section II, which is subsequently sent to the server for further optimization.

Meanwhile, machine learning algorithm developers can construct their own models and share them with others on the same platform. This collection of well-known machine learning algorithms is referred to as the "model zoo." A model zoo comprises model definitions and trained model parameters, essential for using the model in predictions and other analytical tasks. Notably, the backend of the model zoo corresponds to the "step zoo" of COULER , as each model runs as one step in a workflow. Therefore, the GUI and related actions align with COULER 's programming. Leveraging the interactive GUI, a significant portion of workflows (e.g., over 60%) in the cluster are executed via the GUI, addressing the rapid development needs of machine learning applications.

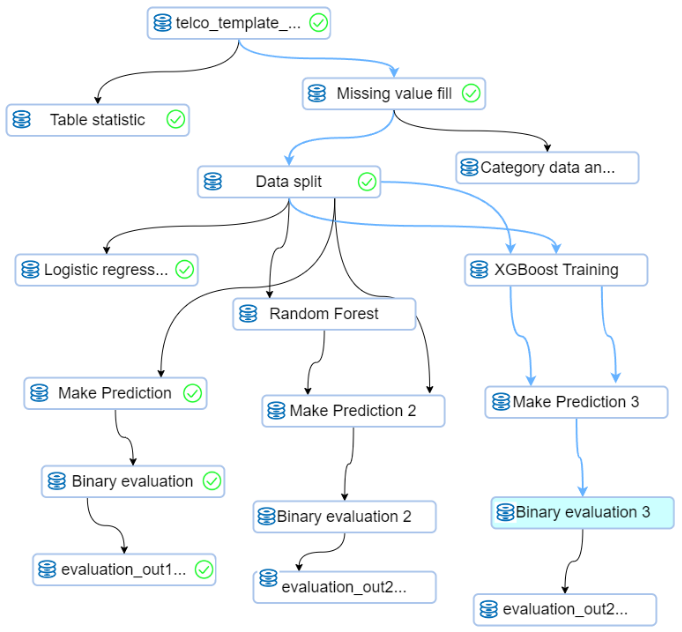


Fig. 10: GUI for the workflow

E. SQL and SQLFlow

In addition to the GUI and Python programming interface, SQLFlow [45] offers an SQL-like language to train machine learning models and employ the trained models for predictions. COULER serves as the default backend for SQLFlow [45]. All optimizations discussed in this work aim to enhance SQLFlow's model training speed. Typically, a SQLFlow SQL statement is converted into Couler programming code, which then initiates a workflow in Kubernetes. An example can be seen in code B-E, where a DNNClassifier model is trained using TensorFlow Estimators [14] on the sample data, Iris.train.

```sql
SELECT *
FROM iris.train
TO TRAIN DNNClassifier
WITH model.n_classes = 3,
model.hidden_units = [10]
COLUMN sepal_len, sepal_width,
    petal_length
LABEL class
INTO sqlflow_models.my_dnn_model;
```

Based on the trained model above, the user can submit a SQL query to get the validation data, then apply the trained model to make a prediction B-E over the selected data. The output of SQL is the data with the prediction value. Naturally, users also could start a data analysis job over the predicted results based on SQL.

```sql
SELECT *
FROM iris.test
TO PREDICT iris.predict.class
```

```
USING sqlflow_models.my_dnn_model;
```

APPENDIX C
RUNNING EXAMPLE OF NL TO UNIFIED PROGRAMMING INTERFACE

We illustrated an example in Figure 11, displaying the full automated process of converting Natural Language to Unified Programming Coded. The goal of this example is to choose the optimal image classification model from ResNet, ViT, and DenseNet, by showcasing the transformation from natural language descriptions to code generation.

Step 1: Modular Decomposition

Initially, we employed a *chain-of-thought* strategy to break down the original natural language descriptions into smaller, more concise task modules. For the given workflow description: *I need to design a workflow to select the optimal image classification model...*. Through this strategy, we identified the following task modules: Data Loading, Model Application (ResNet, ViT, DenseNet), Model Training, Model Validation, Model Comparison, and Model Selection.

Step 2: Code Generation

For each independent task module, we leveraged Large Language Models (LLMs) to generate code. Given each module has a clear and singular task, this enhances the accuracy and reliability of the generated code. For instance, for the *Model Training* task, we generated the relevant code related to training models, ensuring all models use the same training and validation datasets.

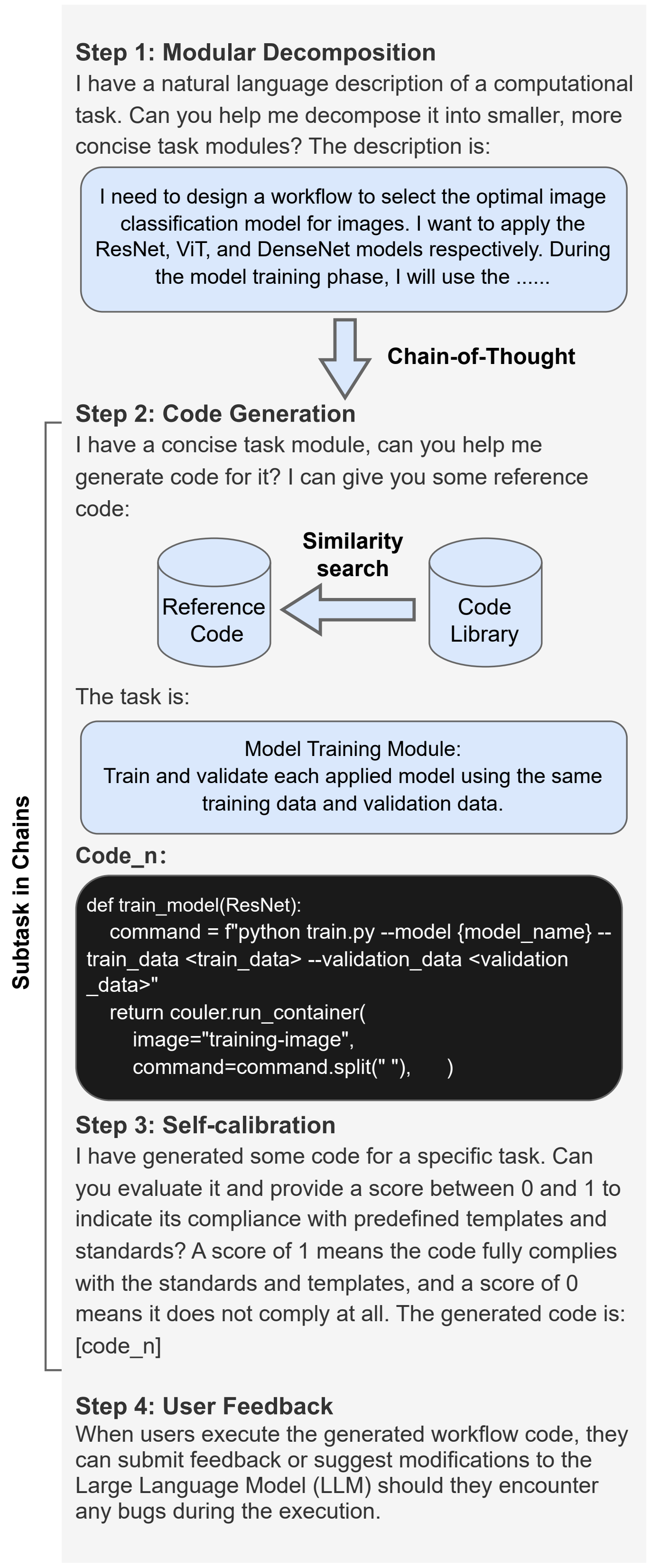
Step 3: Self-calibration

Subsequently, we incorporated a self-calibration strategy to optimize the generated code. This strategy offers improvement suggestions by comparing the generated code with predefined templates in terms of similarity, and these suggestions can be automatically applied to refine the code further, ensuring its compliance with COULER norms and standards. For example, the generated code for model training was compared and optimized against a predefined training code template as necessary.

Step 4: User Feedback

Finally, users have the opportunity to review and validate the generated COULER code. If it does not meet users' requirements, they can provide feedback and suggestions. The system will utilize this feedback to optimize the generated code, enhancing the precision of code generation in future tasks. For instance, if users find the model comparison methodology to be insufficient or biased, they can suggest modifications, allowing the system to adjust the code accordingly based on user feedback.

Through these steps, we not only transformed natural language descriptions into executable code but also ensured the precision, consistency, and efficiency of the generated code. This enables users with limited programming experience to easily realize their computational tasks and workflow needs.



Step 1: Modular Decomposition

I have a natural language description of a computational task. Can you help me decompose it into smaller, more concise task modules? The description is:

I need to design a workflow to select the optimal image classification model for images. I want to apply the ResNet, ViT, and DenseNet models respectively. During the model training phase, I will use the ......

Step 2: Code Generation

I have a concise task module, can you help me generate code for it? I can give you some reference code:

The task is:

Model Training Module:
Train and validate each applied model using the same training data and validation data.

Code_n:

```
def train_model(ResNet):
    command = f"python train.py --model {model_name} --train_data <train_data> --validation_data <validation_data>"
    return couler.run_container(
        image="training-image",
        command=command.split(" "),        )
```

Step 3: Self-calibration

I have generated some code for a specific task. Can you evaluate it and provide a score between 0 and 1 to indicate its compliance with predefined templates and standards? A score of 1 means the code fully complies with the standards and templates, and a score of 0 means it does not comply at all. The generated code is: [code_n]

Step 4: User Feedback

When users execute the generated workflow code, they can submit feedback or suggest modifications to the Large Language Model (LLM) should they encounter any bugs during the execution.

Fig. 11: Running Example: NL to Unified Programming Code