Couler: Unified Machine Learning Workflow Optimization in Cloud

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

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00393

***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, LLM, 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 [20], [32]. 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. Mingjie Tang is the corresponding author.

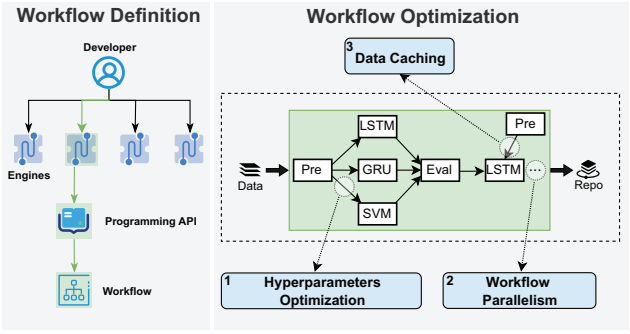


Fig. 1: An example of a financial company's journey in leveraging machine learning to predict market trends.

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 con-

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00393

struct 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 [2], Tekton Pipelines [35], 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 [9], [27], [33], code generation from natural language descriptions [22], [42] and database performance tuning [17]. 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, 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.

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

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

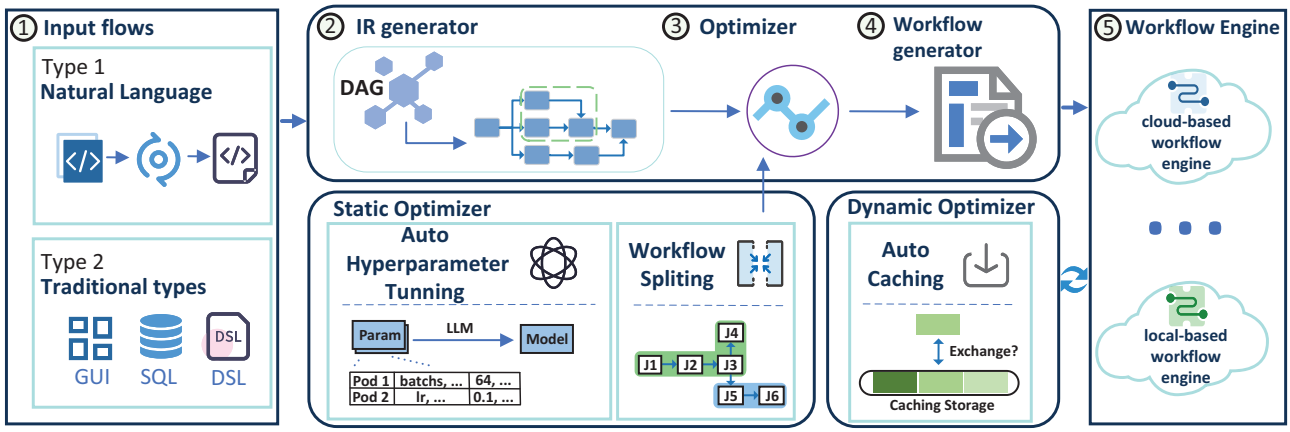


Fig. 2: Overview of COULER Architecture.

functions such as executing scripts, containers, or jobs, stipulating conditions, and managing multiple instances of a job, among others. For example, the code 1 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 Appendix A of our tech report [34].

```
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")]
    ])
diamond() /*Execute the diamond function.*/
submitter = ArgoSubmitter()
/*Submit and run the workflow over Argo.*/
couler.run(submitter=submitter)
```

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 [39] 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 automatically. They provide a structured approach to ensure the precision and correctness of the code generated.

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

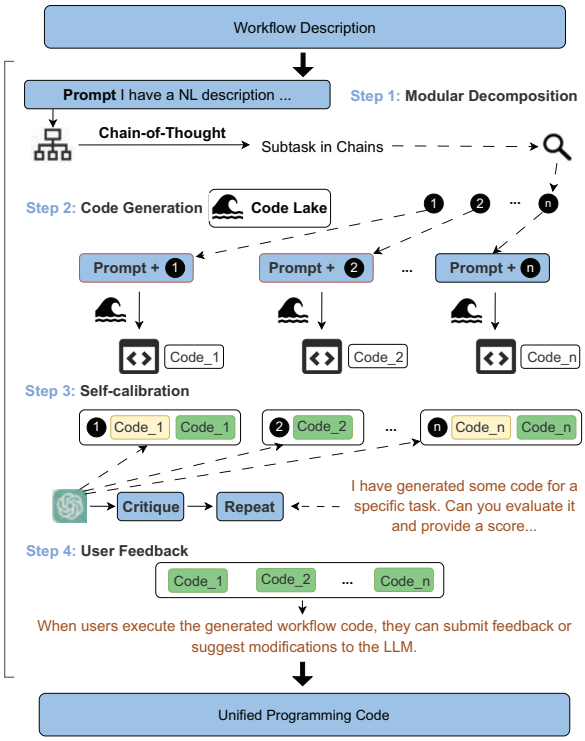


Fig. 3: NL to Unified Programming Interface

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

Step 3: Self-calibration: After generating the code for each subtask, we integrate a self-calibration strategy [37] to optimize the generated code. This strategy evaluates the generated code by having LLMs critique it, as shown in Algorithm 1. In line 8, there may be complex scenarios in which achieving the desired score is impractical for various reasons. Users can adjust $Baseline\_Score$ in instances where it is set too ambitiously, rendering it unattainable. Initially, we define a

baseline score S_b as the standard evaluation 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 in textual format. 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

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

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 .

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 [18]. 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 [44]. 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 simplify 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 formulate 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. Maximizing 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 α and β are used to adjust the weights among the three factors: reconstruction cost, reuse value, and cache cost. As the impact of these factors on efficiency varies in different training scenarios, it is necessary to adjust them according to the actual situation.

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. We will recompute the caching importance factor of all remaining items in the Caching Storage whenever an item is removed. 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

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**

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.

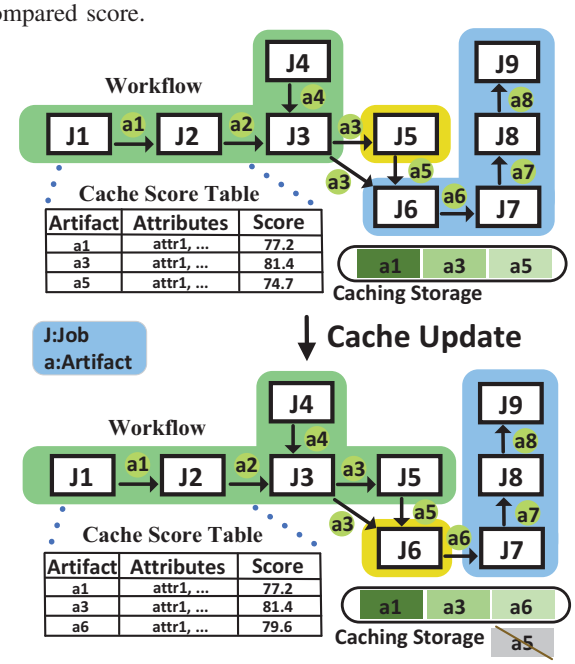


Fig. 4: Running Example of Automatic Caching

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 $a6$.

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 ← ∅, W_s ← ∅
2: markUnVisited(G)
3: for all n_i ∈ N do
4:     if not Visited(v_i) then
5:         NodeSelection(n_i, G, N_c, C_u, C_t)
6:     end if
7: end for
8: function SplitWorkflow(v_1, C, G, W_s, Cand)
9:     b_i ← BudgetOnUnVisitedVertex(G)
10:    if (b_1 ≤ C) then
11:        W_s ← W_s + G
12:        return W_s
13:    end if
14:    MarkVisited(v_i)
15:    C̄ ← Cand + v_1, b_2 ← BudgetOnGraph( C̄)
16:    if b_2 ≥ C then
17:        W_s ← W_s+ Cand, Cand ← v_1
18:    else
19:        Cand ← C̄
20:    end if
21:    for all v ∈ adj(v_1) do
22:        if not Visited(v) then
23:            SplitWorkflow(v, C, 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. For example, α exceeds 2 MB or β exceeds 200. Naturally, if a workflow is bigger than a predefined budget, it needs to be split into small ones.

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 [14] 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 depth-first search (DFS) over a DAG. Algorithm 3 goes through each vertex of the graph and puts 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 recursively 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 [10] and model information from Model Card [19]. 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 $\mathcal{H}$ [46]. 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 Appendix C of our tech report [34].

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. Regarding the expressiveness of Couler 's API compared to the complete APIs of the supported workflow engines, Couler has achieved over 90% coverage of the Argo API. Additionally, it supports approximately 40-50% of the Airflow API. We are actively working to enhance our support for Airflow and other workflow engines. Due to space constraints, we put extensive discussions on implementation of Couler into the Appendix B of our tech report [34].

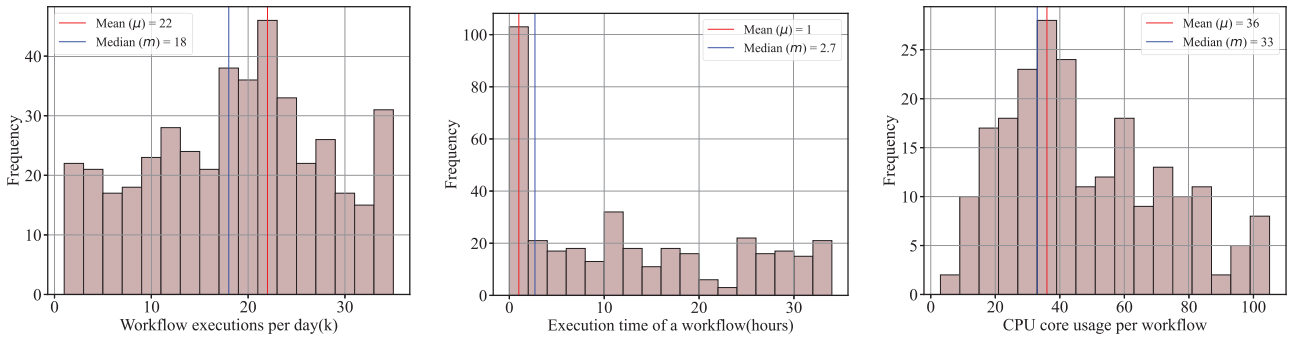
VI. Evaluation

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?

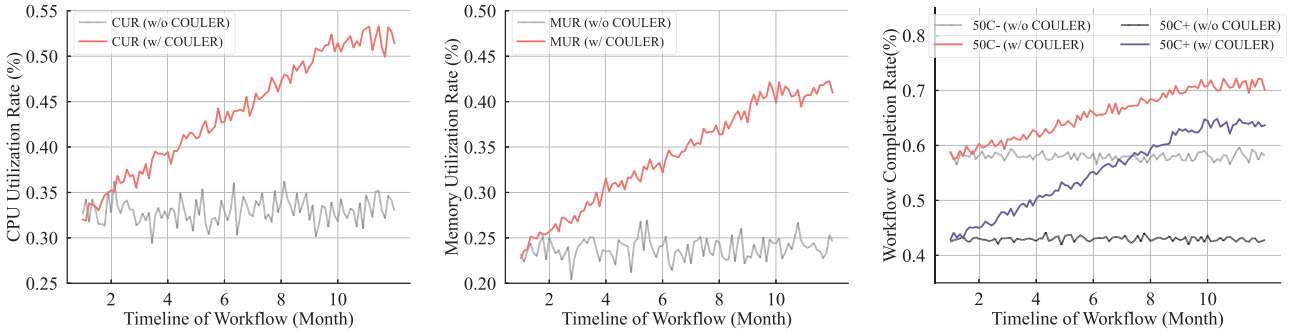
A. Experiment Setup

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



(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

workflow operations provides accurate statistical estimates of the actual gain.

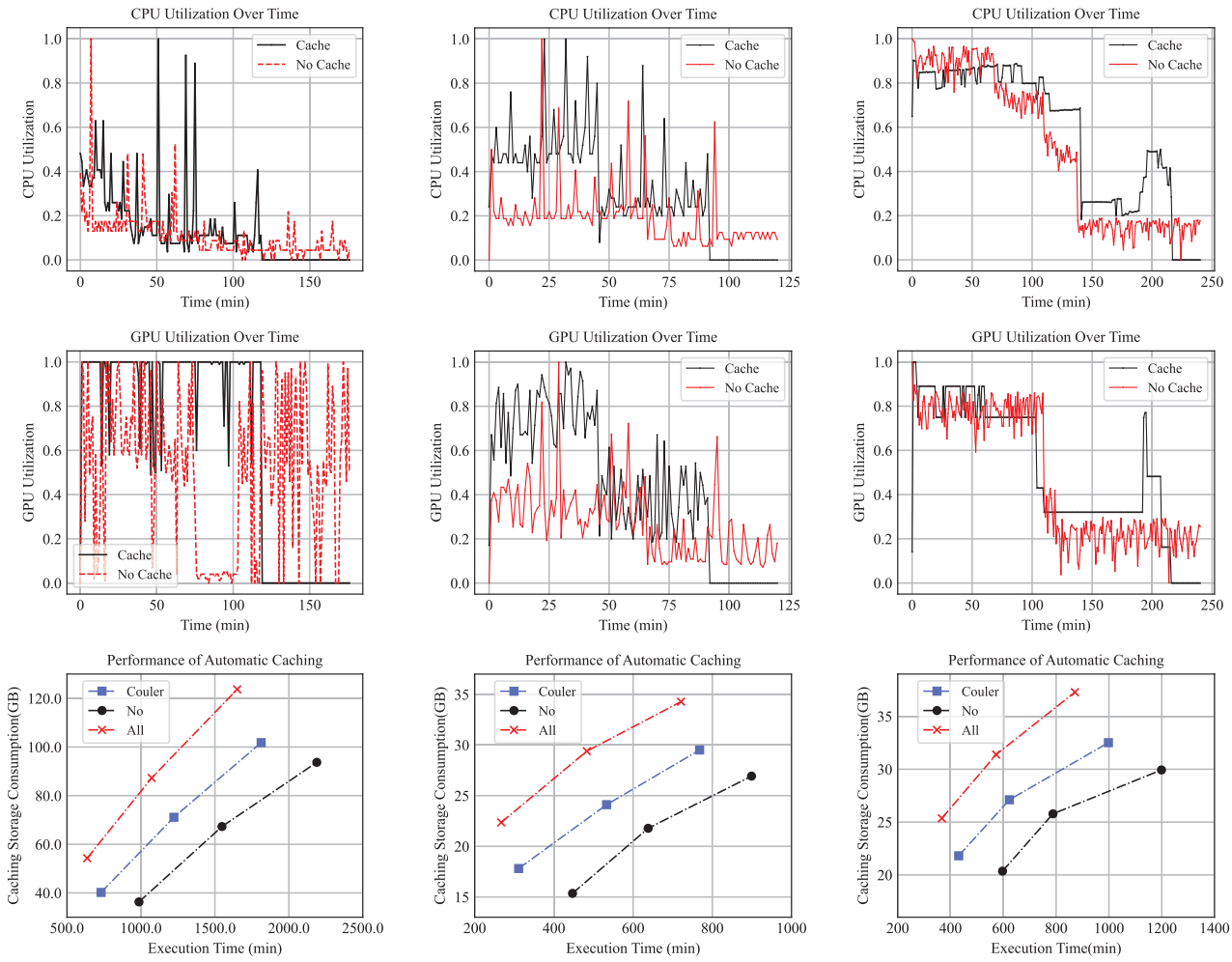
Workload. We design a multi-modal workflow in an isolated production environment to minimize interference of the production environment and conduct a more comprehensive assessment of COULER 's capabilities. By selecting appropriate component containers for model training, we evaluated the system's caching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

Production insights. Our work is motivated by previous research conducted within Google GCP [41], which highlighted substantial computational waste in ML workflows. Building on these findings, our contributions are diverse, encompassing simplicity and extensibility, automation, efficiency, as well as real-world impact and adoption. We believe the real-world



(a) Multimodal Training (b) Image Segmentation (c) Language Model Fine-tuning

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

adoption and application of our system by ANT GROUP and other organizations attest to its practicality and efficacy in production settings. Due to space constraints, we put extensive discussions on production insights into the Appendix E of our tech report [34].

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 five 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. (4) **FIFO**, first in first out; (5) **LRU**, least recently used. Based on empirical experience, we choose $\alpha = 1.5$ and $\beta = 1$ in these experiments for equation 6.

Figure 7 illustrates the variations in CPU and GPU usage over time, comparing COULER 's caching strategy with other caching strategies. Due to space constraints, detailed experimental results for FIFO and LRU can be found in

Appendix D.A of our tech report [34]. It is evident that employing COULER 's caching strategy enhances GPU and CPU utilization, allowing the entire process to complete in less time. This is because that automatic cache mechanism can reduce the frequency of I/O operations. And according to existing work [4], [26], this reduction in I/O overhead is significant as it can substantially decrease the time wasted on these operations, leading to a more efficient workflow execution. 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. COULER 's strategy tends to conserve resources by avoiding unnecessary caching, yet still reaps the performance benefits of caching the most impactful intermediate results. We also calculate the cache hit ratio of the COULER caching strategy, which, under reasonably set parameters, averages 84.21% in production environments, significantly improving the efficiency of workflow execution.

2) Performance study with Data caching: In this section, we investigate the impact of caching on data reading performance by first examining the effect of table caching using two tables from an ads recommendation application, highlighting how caching enhances data loading and deep learning model training efficiency on a hybrid cluster. Secondly, we assess the caching performance for reading small and big files stored remotely, demonstrating significant improvements in data reading speed through local caching. Due to space constraints, we put detailed discussions into the Appendix D.C of our tech report [34].

3) Performance Study with Cache Sizes: We further designed experiments to analyze the impact of different cache sizes on COULER 's performance. In the same three scenarios: Multimodal Training, Image Segmentation, and Language Model Fine-tuning, we set the available cache sizes to 10G, 20G, and a more ample 30G, respectively, and recorded the resource utilization and execution time of the workflows under these conditions. Detailed experimental results can be found in Appendix D.B of our tech report [34]. Analysis shows that, under limited cache size conditions, COULER can still effectively improve the efficiency of workflow execution, but its effectiveness increases with the size of the cache.

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 [22]. The 'pass@k' metric is a widely used evaluation method in code generation models. It assesses the model's capacity to produce accurate code within its top 'k' predictions. A higher 'pass@k' percentage indicates the model's reliability in generating correct code options without the need for additional inputs or iterations. 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 [22].

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 provided an example that illustrates the entire process of converting natural language into COULER code. It demonstrates the generation of syntactically correct code by the LLM. This example aims to select the best image classification model among ResNet, ViT, and DenseNet by showing the transformation from natural language descriptions to code generation. The details are presented in Appendix C of our tech report [34].

3) Cost Analysis and Future Prospects Discussion: On the matter of cost, we understand the concerns about the economic feasibility of deploying LLMs, especially considering the cost per token for each workflow. So we present the average costs for each workflow in terms of the number of tokens processed by LLMs and the corresponding money for model "GPT-3.5-turbo" and "GPT-4" in Table III. COULER has shown promising results for a range of tasks except for some complex workflows. The use of LLMs in our research is primarily aimed at exploring the potential of these models to streamline and enhance the code generation process.

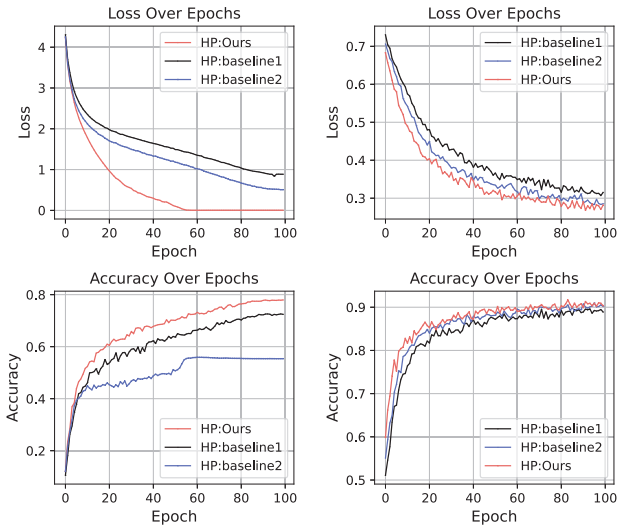
TABLE III: Cost Analysis of Workflow Generation

Cost / Workflow	**Workflow Generation**	
	GPT-3.5-turbo	GPT-4
Token	3212.1	3813.7
Money ($)	0.005	0.140

We acknowledge that current LLMs already demonstrate a satisfactory level of accuracy in code generation. Indeed, we are considering fine-tuning as a viable method to enhance the quality of the generated code. Specifically, our team has conducted work on Multi-LoRA optimization for fine-tuning [43]. Additionally, we are developing a workflow for code generation by fine-tuning 'llama2'. We will soon release a fine-tuned model in the COULER open-source repository.

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,



(a) Auto Configuration for CV (b) Auto Configuration for NLP

Fig. 8: Effect of Auto Hyperparameter Configuration

we apply automated hyperparameter tuning to both the *cv* and *nlp* modules, with *HP:Ours* as our recommended parameter. As depicted in Figure 8, the recommended parameters exhibit the lowest loss and the highest accuracy, showcasing the potent capability of COULER 's Automatic Hyperparameter Tuning. HP-baseline1 represents a set of hyperparameters that were manually selected based on expert knowledge and empirical best practices in the field. HP-baseline2 corresponds to a set of hyperparameters derived from historical benchmarks and recommendations in the literature.

F. Comparative Learning Analysis of Workflow Engines

To assess the effectiveness of our system's unified programming model, we conducted a survey, where we asked 15 engineers who were not familiar with these workflow engines to learn and use code snippets with similar functionality in COULER , Argo, and Airflow. We measured the time it took for them to understand and work with the provided code samples. The results of this survey in Table IV clearly indicate that our COULER API is more user-friendly and easier to learn.

TABLE IV: Workflow Learning Comparative Analysis

Time / Workflow	**Workflow Engines**		
	COULER	Argo	Airflow
Time (min)	18	61	50

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 [3], [16], [25], [29]. Kubernetes [28] 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. It supports various big-data frameworks (e.g. Apache Hadoop MapReduce [11], [31], Apache Spark [44], Apache Kafka [13], Apache Flink [8] etc). More recently, Kubeflow [16] allows users to submit distributed machine learning tasks on Kubernetes.

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 [3], [16], [25]. Kubernetes [28] boasts a rapidly growing community and ecosystem, providing robust support for workflows. TFX [20] 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 [45].

Workflow Engine. A workflow engine is a software application that manages business processes. Argo Workflows [2] 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 [23] is a workflow engine in the Hadoop ecosystem. Kubeflow [16] 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 [40]. There are also recently workflow engines like Ray [21], CodeFlare [5] and ThunderML [30]. COULER is inspired by the design of PyTorch [24], which compiles a high-level AI model to a DAG.

Automated Machine Learning. Automated Machine Learning (AutoML) [12], [38] 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 [15], [36], AutoSklearn [7], AutoGluon [6], and Auto-PyTorch [47], alongside commercialized frameworks.

Workflow Optimization and Query Optimization. Workflow optimization applies broadly, covering areas such as scientific workflows, business processes, and cloud computing, and addresses tasks beyond data processing, including computational and data movement activities. It also faces unique challenges like deadlines, budget constraints, and fault tolerance, which are less common in query optimization. Additionally, workflow optimization utilizes specific strategies like dynamic scheduling, partitioning, and machine learning for predictive optimization, highlighting its distinct requirements compared to the more static nature of query optimization.

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. Furthermore, COULER boosts computational efficiency through automated caching, large workflow auto-parallelization, and hyperparameter tuning.

REFERENCES

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

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

[3] 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," in *mlflow*, ser. DEEM'20. New York, NY, USA: Association for Computing Machinery, 2020.

[4] S. W. Chien, A. Podobas, I. B. Peng, and S. Markidis, "tf-darshan: Understanding fine-grained i/o performance in machine learning workloads," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 359–370.

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

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

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

[14] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, p. 558–562, Nov. 1962.

[15] 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.

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

[17] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang, "Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization," *arXiv preprint arXiv:2311.03157*, 2023.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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. 17–28, Dec. 2018.

[26] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable deep learning via i/o analysis and optimization," *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 2, pp. 1–34, 2019.

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

[28] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472: k8s, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[29] B. Sang, S. Gu, X. Zhan, M. Tang, J. Liu, X. Chen, J. Tan, H. Ge, K. Zhang, R. Ruan *et al.*, "Cougar: A general framework for jobs optimization in cloud," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 3417–3429.

[30] 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.

[31] 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.

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

[33] 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.

[34] "Tech report of couler: Unified machine learning workflow optimization in cloud," https://arxiv.org/pdf/2403.07608.pdf, Oct. 2023.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] "Awesome workflow engines," https://github.com/meirwah/awesome-workflow-engines, Oct. 2023.

[41] D. Xin, H. Miao, A. Parameswaran, and N. Polyzotis, "Production machine learning pipelines: Empirical analysis and optimization opportunities," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2639–2652.

[42] 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.

[43] Z. Ye, D. Li, J. Tian, T. Lan, J. Zuo, L. Duan, H. Lu, Y. Jiang, J. Sha, K. Zhang *et al.*, "Aspen: High-throughput lora fine-tuning of large language models with a single gpu," *arXiv preprint arXiv:2312.02515*, 2023.

[44] 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.

[45] A. X. Zhang, M. Muller, and D. Wang, "How do data science workers collaborate? roles, workflows, and tools," 2020.

[46] S. Zhang, C. Gong, L. Wu, X. Liu, and M. Zhou, "Automl-gpt: Automatic machine learning with gpt," *arXiv preprint arXiv:2305.02499*, 2023.

[47] 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.