Technical Report of Couler: Optimizing Machine Learning Workflows in Cloud

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

I. 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 1 and code 2), the other way is explicitly defining a workflow as code 3. The main difference is whether DAG is described in the program. The core functions of Couler are listed in Table I, 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 I: API Summary of Couler

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

Code 1: Basic workflow and artifact definition 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 II.

Take the code 1 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():
```

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 II: Artifact support in COULER

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

Code 2: 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 2 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 3: 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 3 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 4 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
4      result = "heads" if random.randint(0,
           1) == 0 else "tails"
5      print(result)
6
7
8  def flip_coin():
9      return couler.run_script(
10         image="alpine3.6",
11         source=random_code)
12
13 # Stop flipping coin until the outputs of
       'flip_coin' is not 'tails'
14 couler.exec_while(couler.equal("tails"),
15     lambda: flip_coin())
```

Code 4: 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),
       model_path)
```

Code 5: 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 5 implements a model searching procedure for a deep learning model (e.g., wide and deep model [3]. 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":
           "binary:logistic"}
10     train_params = {"num_boost_round": 10,
11         "max_depth": 5}
12
13     return xgboost.train(
14         datasource=train_data,
15         model_params=model_params,
16         train_params=train_params,
17         image="xgboost-image",
18     )
19
20 def train_lgbm():
21     train_data = Dataset(
22         table_name="pai_telco_demo_data",
23         feature_cols="tenure,age,
24             marital,address,ed,employ",
25         label_col="churn",
26     )
27
28     lgb = LightGBMEstimator()
29     lgb.set_hyperparameters(num_leaves=63,
30         num_iterations=200)
31     lgb.model_path = "lightgbm_model"
32     return lgb.fit(train_data)
33
34 couler.concurrent([lambda: train_xgboost(),
35     lambda: train_lgbm()])
```

Code 6: 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 6 shows how to choose a best model from two machine learning model (e.g., XGBoost [1] and LightGBM [4]), 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.

II. 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

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

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 7. This CRD 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 7: 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 1 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 **??**, 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.

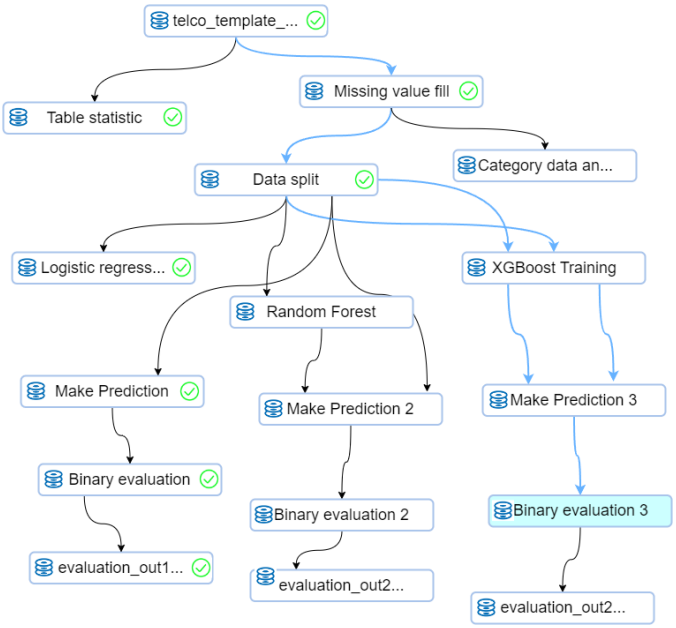

Fig. 1: GUI for the workflow

E. SQL and SQLFlow

In addition to the GUI and Python programming interface, SQLFlow [5] 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 [5]. 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 II-E, where a DNNClassifier model is trained using TensorFlow Estimators [2] on the sample data, Iris.train.

```
SELECT *
ROM iris.train
TO TRAIN DNNClassifier
WITH model.n_classes = 3,
model.hidden_units = [10]
COLUMN sepal_len, sepal_width, petal_len
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 II-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.

```
SELECT *
FROM iris.test
TO PREDICT iris.predict.class
USING sqlflow_models.my_dnn_model;
```

III. Running Example of NL to Workflow

We illustrated an example in Figure 2, 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.

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.

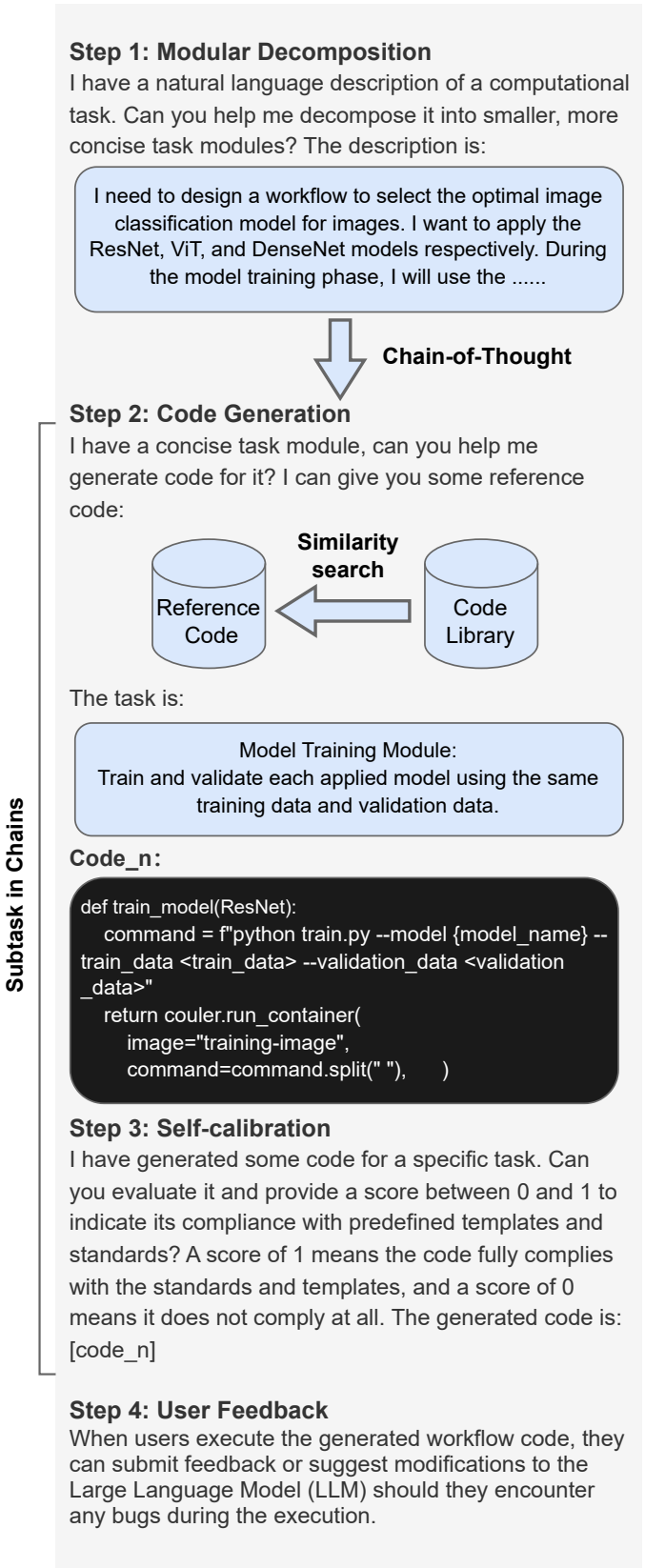

Fig. 2: Running Example: Automatic Code Generation

REFERENCES

[1] "Xgboost," https://github.com/dmlc/xgboost, Jul. 2022.

[2] 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.

[3] 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.

[4] 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. 3149–3157.

[5] 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.