

FlexiFlow: Bandit-based Model Switching in ML Workflows

Abhilash Jindal
ajindal@cse.iitd.ac.in
IIT Delhi
India

Tanu Malik
tanu@missouri.edu
University of Missouri
USA

Anshik Sahu
cs1210577@cse.iitd.ac.in
IIT Delhi
India

Todd Nief
tnief@uchicago.edu
University of Chicago
USA

Shankaradithyaa V
shankaradith@iisc.ac.in
IISc Bangalore
India

Amitabh Chaudhary
amitabh@uchicago.edu
University of Chicago
USA

Bhanu Prakash Vangala
bv3hz@missouri.edu
University of Missouri
USA

Tvisha Malik
cs1210085@cse.iitd.ac.in
IIT Delhi
India

Aaron Schein
schein@uchicago.edu
University of Chicago
USA

Abstract

Model optimizations help improve inference performance and accuracy of ML workflows. However, relying on a single model to perform inference across all data batches often fails to maximize accuracy and thus overall performance. In many cases, alternate models could perform better on specific subsets of data where a primary model underperforms. Our experiments with real ML workflows indeed show that switching models improves workflow accuracy by up to 23%.

Yet, current systems lack the ability to adaptively switch between models based on performance, forcing users to manually test models in sequence. We present FlexiFlow, a dataflow system that dynamically switches between alternate models when the current model exhibits low accuracy. FlexiFlow learns to rank models using a novel multi-armed bandit approach that accounts for model runtimes, probability of passing user-defined assertions, and the computational structure of the ML workflow. We show that the standard Thompson sampling approach is insufficient for switching models in ML workflows. In contrast, our proposed approaches are effective and scales to complex real-world ML workflows. Experiments show that switching models at runtime while reusing intermediate results provides higher accuracy, but also 48% efficiency gain compared to sequential workflow runs.

1 Introduction

With the increasing use of machine learning in production systems, engineers often encounter practical problems in deploying and maintaining ML models and workflows. Traditionally, models are often learned in notebooks or scripts, serialized (i.e., “pickled”), and embedded within web servers—*a.k.a* deployed—to expose prediction APIs. However, deployment often reveals a range of issues, particularly related to model performance and accuracy.

A recurring concern is the issue of soft failures—situations where the model does not crash but delivers incorrect or degraded predictions, often due to factors like data drift [16]. These soft failures compromise prediction accuracy and, in turn, the overall effectiveness of the deployed system. When workflows consist of multiple steps and models, such degradations must be promptly identified



Figure 1: ID Card ML workflow.

and addressed. Developers commonly rely on trace logs to identify problematic data batches, but exhaustively testing alternative model configurations is resource-intensive and time-consuming.

Consider a deployed ML workflow, visualized in Figure 1, for identity verification in KYC (Know Your Customer) scenarios using identification cards (e.g., driver’s licenses, passports). The workflow processes an image or scan of the ID card and outputs a structured record with fields such as name, date of birth, ID number, and expiration date. Each step in this pipeline—such as image cropping, color channel selection, OCR, card classification, filtering, and field extraction—may be implemented using different ML models. For instance, the cropping step could use models like YOLO [38], SSD [32], or Faster R-CNN [39], while OCR could be performed by tools such as Tesseract, EasyOCR, Google Vision API, AWS Textract, or Microsoft Read API. In practice, only one *single* model is typically selected for each step based on evaluation over training data. However, this static selection often fails to maintain high accuracy across all real-world inputs during deployment. If a single model underperforms, the entire workflow may need to be re-executed from the beginning with an alternative model, significantly increasing the management burden and latency.

One approach to mitigating soft model failures on specific inputs is to dynamically switch between models based on their performance. For example, in the OCR step of an ID card processing pipeline, if EasyOCR fails on a given input, the system can switch to an alternative like Tesseract. However, this requires the system to assess relative model performance—not just using aggregate accuracy metrics, but by detecting soft failures (e.g., invalid outputs or low confidence) on individual inputs. Even when estimates of model performance are known, a key challenge remains: determining the optimal order in which to try models to minimize latency while maintaining correctness.

Recently, these challenges have been addressed, albeit independently, in the machine learning domain. Model assertions [28] provide a data-independent way to assess model performance. In the context of ID card processing, such assertions might include checks like validating that the extracted date of birth is a legitimate calendar date, ensuring the age is under 120 years, or applying confidence thresholds to predictions. Separately, explore-exploit strategies [42] offer a principled method for determining which configurations to use first. These approaches balance exploiting configurations that have performed well historically with exploring new alternative configurations, adapting dynamically based on observed rewards. To our knowledge, no existing ML dataflow system integrates both model assertions and explore-exploit strategies to guide model switching decisions. Integrating model assertions with explore-exploit strategies requires a non-trivial coordination of correctness and efficiency, as the system must adapt to assertion violations while maintaining acceptable performance.

In this paper, we present FlexiFlow a dataflow system for multi-step ML workflows that learns to perform model selection and switching when models (soft) fail on assertions. FlexiFlow allows a user to configure each step of an ML workflow with multiple model choices. To determine the best model configuration, *i.e.*, combination of model choices at each step, we equip FlexiFlow with a Bayesian statistics-based optimizer that samples from the posterior reward distributions of model choices, enabling the estimation of the most effective model configuration for a given input. If a model configuration still fails assertions, to explore alternative configurations, FlexiFlow rolls back to a previous model operator and efficiently tests a sequence of new configurations. We demonstrate that a variety of explore-exploit Bayesian strategies are possible within the runtime to enhance performance. Our findings indicate that while the space of all possible workflow configurations can be large, Bayesian strategies can quickly find good configurations.

1.1 Our Contributions

Figure 2 shows the different choices and assertions specified for an instance of the ID card ML workflow. In the ChannelPicker step, five color channel choices are made available: all channels (all), blue only (b), green only (g), red only (r), and a combination of blue and green channels (bg). In the OCR step, the user provides choice between two engines: Tesseract and EasyOCR. Additionally, the workflow allows for different classifier choices in the classifier step. The figure also depicts assertion checks on extracted data. As shown, there are a total of 20 different workflow configurations, corresponding to 5, 2, and 2 operator choices at various steps. The configuration space grows quickly as the number of choices increases or as more steps are added.

Although an ML workflow might select models or configurations based on performance during training, these choices often degrade when deployed, due to shifts in the data distribution. To remain effective, the system must adapt to changing conditions at inference time. This motivates the use of online learning approaches where, rather than training the model selection process all at once on a fixed dataset, the system incrementally updates its model selection process with the arrival of each new example or batch of examples.

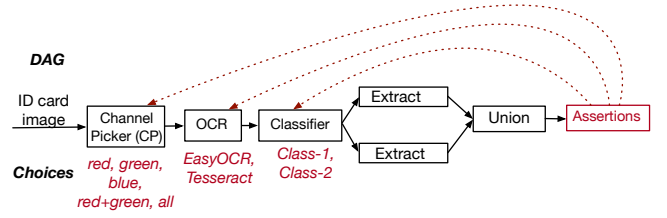


Figure 2: ID Card workflow in FlexiFlow. Choices for each workflow step are indicated below in red italics. Soft failure of assertions necessitates switching (dashed arrows) between model choices.

A common approach to online learning is the multi-armed bandit (MAB), where the goal is to choose among k configurations by balancing exploration and exploitation to maximize expected reward. However, FlexiFlow departs from the standard MAB setting in two key ways:

First, while there are k overall workflow configurations, each configuration is composed of multiple steps (or operators), and each step has its own set of model choices. This implies that a distinct MAB problem exists at each stage of the workflow. This MAB problem likely includes interactions across the decision at each step, resulting in a bandit problem that does not fit neatly into the standard contextual or hierarchical bandit structure.

Second, unlike traditional MAB settings where each trial is independent, in FlexiFlow, if a given configuration fails (e.g., violates a model assertion), it should not require restarting the entire workflow. Instead, the system should be able to ‘rollback’ to the point of failure and selectively modify only the relevant operator choice. For instance, as shown in Figure 2, if a classification step yields incorrect results, the system should be able to switch classifiers without re-running upstream components like OCR or preprocessing.

These distinctions highlight the need for a bandit-based approach that leverages the computational structure of the workflow graph, enabling localized reconfiguration and efficient exploration of the model choice space for improved overall performance. We make the following contributions:

- **A novel programming model with operator choices and assertions:** We introduce primitives within an existing dataflow language to support the expression of both operator choices and assertions. The language runtime, to support multiple choices, incorporates an adaptive rollback mechanism that reverts to specific operators when an assertion fails. This rollback mechanism is guided by an optimizer (described next), allowing for rollbacks to intermediate nodes within the workflow and not just the root step. To facilitate incremental computation after rollback, we leverage past work on dataflow error handling [27].
- **Bayesian strategies for automatic error correction:** We propose two multi-armed bandit-based strategies to learn an optimal sequence of configurations:
 - The first configuration strategy uses Thompson sampling with posterior distributions over model configurations, balancing exploitation of high-payoff configurations that pass

the most assertions with exploration of alternative configurations. Execution time is explicitly modeled as part of the optimization strategy.

- The second hybrid strategy exploits the structure of the choice-based programming model by running separate multi-armed bandit optimizations at each workflow step, tracking cumulative payoffs for configurations and updating posteriors for operator choices on high-payoff model configurations.

Using microbenchmarks and real applications, we show that the hybrid (second) strategy provides a learning advantage.

- **A research prototype:** We present FlexiFlow, an experimental Python prototype that supports efficient rollbacks, incremental computation, and the proposed language extensions and optimization strategies. We evaluate FlexiFlow with three types of ML workflows: document processing, image classification, and question-answering. For each workflow, we describe its formulation in FlexiFlow, the range of choices and assertions, and the performance of our proposed strategies. Our key findings are:
 - Adding operator choices in workflows can improve F_1 scores by upto 23% compared to workflows without them. We find that this is because different configurations, with different ML models, often have uncorrelated failures.
 - It is imperative for strategies to learn runtimes of operator choices along with their probabilities of passing assertions. Further, by making them aware of the rollback approach of FlexiFlow, they can quickly learn good configurations.

The rest of the paper is organized as follows: Section 2 presents the FlexiFlow architecture and describes the programming model consisting of operator choices, assertions, and rollbacks. Section 3 formulates the configuration selection learning problem in terms of assertions and rollbacks. In Section 4 we describe Bayesian strategies that estimate the probability of choosing a configuration with the highest reward. In Section 5 we present our implementation of FlexiFlow and evaluation on microbenchmarks and four real ML pipelines. We discuss our results in Section 6, present related work in Section 7, and conclude in Section 8.

2 The FlexiFlow Execution Model

FlexiFlow is defined within the context of a dataflow language that enables users to express complex machine learning (ML) workflows. We begin by reviewing the fundamentals of such a language and its data model, and then introduce two novel programmatic constructs provided by FlexiFlow: operator choices and workflow assertions.

A basic dataflow language, such as Spark-SQL [5] or SCOPE [50], models an ML workflow as a directed acyclic graph (DAG). In this DAG, nodes represent operators, each of which performs a data transformation; edges represent data dependencies between operators (i.e., the flow of intermediate datasets). Let d denote an input data item (e.g., a row or an image), and let op denote a map transformation function implemented by an operator. Then, the semantics of an operator are defined as:

$$op(d) \rightarrow d'$$

in which d' is the output data item produced by applying op to d . The execution of a dataflow program thus proceeds by propagating

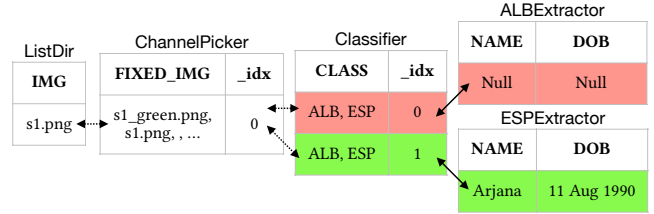


Figure 3: FlexiFlow data model. Some operators are omitted for clarity.

data from source nodes to sink nodes, applying each operator as data becomes available. The runtime supports pipeline parallelism [22]: partial outputs from parent operators can be streamed to child operators without waiting for the entire input to be processed.

Typical operators are data preprocessing operators such as extracting structured rows from a file, processing them, combining rows, or outputting to disk or user-defined operators such as processing video frames or images into structured data. Each operator consumes a well-defined input schema and produces a corresponding output schema as shown in Figure 3. The runtime verifies schema compatibility along edges in the DAG to ensure compositional correctness. In particular, operators are map operators that apply the operator independently to each data item, and currently do not include logical operators such as joins, and aggregates.

FlexiFlow augments this basic model by introducing three core constructs: operator choices and workflow assertions as language constructs, and rollback as a runtime construct. In FlexiFlow, each map operator op may be instantiated by one of multiple alternative implementations. Let $\{op^1, op^2, \dots, op^k\}$ denote the k possible implementations. The operator is now modeled as a non-deterministic transformation, selected at runtime:

$$op^i(d) \rightarrow d'^i, \text{ where } i \in 1, \dots, k$$

That is, for a given input d , the runtime selects an instance choice op^i and produces the corresponding output d'^i . Figure 2 also shows (in italics) multiple choices available for each operator.

In addition to operator choices, FlexiFlow allows users to define assertions on intermediate or final outputs of the workflow. These are Boolean-valued functions $\phi(d') \in \{\text{true}, \text{false}\}$ that act as semantic checks (e.g., validating that a date of birth is valid or that confidence scores exceed a threshold). In general there can be many assertions at different locations of a DAG, but for ease of exposition we assume a single assertion represented as a sink node in a DAG. Assertions provide a signal of correctness and are used by the runtime to detect soft failures and guide, as we describe in the next section, operator re-selection.

Finally, when an assertion fails for a given input, FlexiFlow’s runtime rolls back to an operator. Figure 2 shows that ID card can possibly rollback to Channel Picker, OCR or Classifier operators. We describe further implementation details of how to rollback in Section 5.

Architecture. Figure 4 shows the overall architecture of FlexiFlow with the three constructs: operator choices, assertions, and rollback, which introduce in FlexiFlow the decision of which specific operator to rollback to and which operator choice to pick next, when an assertion fails. To address this, FlexiFlow employs a runtime-aware, rollback-aware, learning-based optimizer that learns from

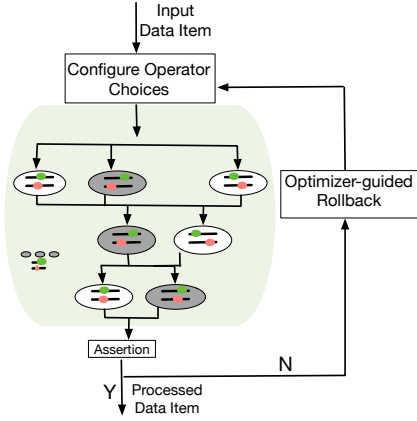


Figure 4: FlexiFlow Architecture. Optimizer maintains estimates for each operator choice and set of operator choices.

past successes and failures at both the operator level and across combinations of operator choices as described next.

3 The FlexiFlow Learning Problem

The choices available at each operator in FlexiFlow multiplicatively increase the space of ML workflow *configurations*.

Let an ML workflow consist of M operators, where each operator i , with $i \in \{1, \dots, M\}$, has k_i possible choices. Then, the total number of workflow configurations is given by:

$$K = \prod_{i=1}^M k_i \quad (1)$$

For example, the ID card processing workflow has $M = 3$ operators—ChannelPicker, OCR, classifier—each having 5, 2, and 2 choices, respectively. One possible configuration within the workflow is [“red”, “Tesseract”, “class 1”]. In total, the workflow has $5 \times 2 \times 2 = 20$ configurations to choose from.

For a given data item, FlexiFlow initially selects a configuration from the set of K configurations, with the hope that the selected configuration passes the assertion. If this selection fails the assertion, the system selects another configuration, continuing until the assertion is satisfied. This process forms a sequence of configurations: $S = (a_1, \dots, a_j)$, $j \leq K$. In the worst case, the sequence can be as long as K , meaning all configurations of an ML workflow must be tested. However, the objective of FlexiFlow is to determine the optimal sequence S^* that minimizes the expected execution cost, as each additional configuration incurs an overhead of re-running the operators.

We observe two properties of any sequence S :

- First, in any sequence of length j , the current configuration a_j is tried if all previous configurations from index 1 to $j - 1$ have failed the assertion.
- Second, if two consecutive configurations share a longer prefix of common operator choices, the rollback execution is faster. Consider the sequence ([“red”, “EasyOCR”, “class 2”], [“red”, “EasyOCR”, “class 1”]) and the sequence ([“red”, “Tesseract”, “class 2”], [“red”, “EasyOCR”, “class 1”]). The first sequence, due to longer

common prefix, leads to a lower re-execution cost, improving overall efficiency.

Given these properties, the goal in FlexiFlow is to identify the best configuration sequence S^* that minimizes the expected runtime, formulated as:

$$\sum_{j \in \{1..K\}} P \left[\bigcap_{i < j} \bar{a}_i \right] \times \mathcal{T}_{a_{j-1} \rightarrow a_j}, \quad (2)$$

in which, $P \left[\bigcap_{i < j} \bar{a}_i \right]$ is the probability that all previous configurations $\{a_1, \dots, a_{j-1}\}$ fail the assertion, and $\mathcal{T}_{a_{j-1} \rightarrow a_j}$ is rollback execution cost from configuration a_{j-1} to a_j .

Expression 2 is of limited practical use, since failure probabilities are unknown for new data items, and expected runtimes in deployment often deviate from training phase estimates due to differences in data distributions. So we next describe how FlexiFlow efficiently identifies S^* from a large space of configuration sequences using an online, Bayesian explore-exploit learning strategy.

4 FlexiFlow’s Bayesian Online Optimizer

In this section, we develop a variety of optimizers, each of which uses a Bayesian explore-exploit learning strategy from the multi-armed bandit framework [42]. The optimizers estimate success probabilities and runtimes to minimize total expected runtimes (Equation 6). Below, we provide a brief background on the Bernoulli bandit problem, followed by FlexiFlow online optimizers.

Background. Consider a scenario where an operator has K possible choices. Each action $a \in \{1, \dots, K\}$ has an unknown success probability θ_k meaning that each time action a is chosen, it yields a binary reward r . r is 1 with probability θ_a (success) and r is 0 with probability $1 - \theta_a$ (failure). The objective of the bandit framework is to learn the success probabilities $(\theta_1, \dots, \theta_K)$ through experimentation, while maximizing cumulative rewards over N trials, where $N \gg K$.

Suppose we have an operator with three possible actions. Lets say over 2005 trials, these actions were selected as follows: Action 1 was chosen 1000 times, yielding 600 successes ($\theta_1 = 0.6$), Action 2 was chosen 1000 times, yielding 400 successes ($\theta_2 = 0.4$), and Action 3 was chosen only 5 times, yielding 2 successes ($\theta_3 = 0.4$, but with high variance due to the small sample size). Figure 5 illustrates the probability density functions (PDFs) for these estimated success probabilities. The variance in the estimates decreases as the number of trials per action increases.

In an ϵ -greedy approach, the learning agent explores by sampling actions uniformly at random with probability ϵ , where ϵ is fixed a priori. During exploitation (with probability $1 - \epsilon$), the agent repeatedly selects the action with the highest estimated mean reward. For example, if action 1 consistently has a higher mean than actions 2 and 3, it will be chosen repeatedly during exploitation, while during exploration all three actions are sampled uniformly.

Allocating an equal number of experimental trials to each action during exploration results in poor performance of an ϵ -greedy approach [40]. Thompson Sampling (TS) fundamentally addresses this limitation of ϵ -greedy by updating the probabilities of success of actions using Bayes’ rule. In Thompson Sampling, each action a has an independent prior belief over each θ_a , which is modeled as

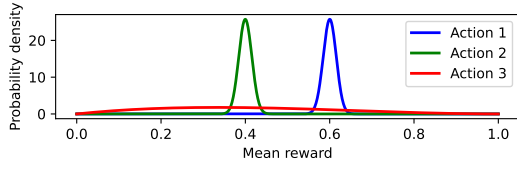


Figure 5: Probability density functions over mean rewards

a Beta distribution:

$$f(\theta_a; \alpha_a, \beta_a) = \frac{1}{B(\alpha_a, \beta_a)} \theta_a^{\alpha_a-1} (1 - \theta_a)^{\beta_a-1} \quad (3)$$

where the beta function $B(\alpha_a, \beta_a)$ is a normalizing constant. As observations are gathered, each action's posterior distribution is updated according to a simple rule:

$$(\alpha_a, \beta_a) \leftarrow \begin{cases} (\alpha_a, \beta_a), & \text{if } x_t \neq a \\ (\alpha_a, \beta_a) + (r_t, 1 - r_t), & \text{if } x_t = a \end{cases} \quad (4)$$

in which r_t represents the actual reward in the trial t^{th} that attempts action x_t . When $\alpha_k = \beta_k = 1$, the prior probability of θ_k is uniform over $[0, 1]$. For each data item, only the Beta parameters (α_a, β_a) of the selected action a are updated based on the observed reward r_t . If the action succeeds ($r_t = 1$), α_k increases, otherwise β_k increases. As the number of trials grows, the distribution mean of the Beta distribution $\frac{\alpha_k}{(\alpha_k + \beta_k)}$, which represents the ratio of success to total trials, approaches the empirical mean of successes. Instead of explicitly alternating between exploration and exploitation phases (as in ϵ -greedy), Thompson Sampling samples from the posterior distribution of each action's success probability θ_a . Actions with higher success probability estimates are sampled more frequently, but uncertainty in estimates naturally encourages exploration.

4.1 FlexiFlow Optimizer for Single Operator with Multiple Choices

Traditional Thompson Sampling (TS) updates success probabilities based on the observed reward from a single action/operator choice at each trial, accumulating cumulative rewards over time. However, FlexiFlow introduces a key difference: when an operator choice fails, alternative choices are repeatedly attempted via rollbacks until a successful configuration is found. These rollbacks introduce additional runtime costs, as each re-execution incurs the overhead of rerunning the operator choice. To address this, we first describe a modified version of Thomson Sampling (TS), termed Operator-level TS (OTS), in which only the reward function is changed and is modeled with expected runtimes of probability of success or failure of the assertions. We consider OTS for a single-operator scenario with multiple choices. We then extend this algorithm to include the computational structure of the workflow in operator and choices along with rollbacks in a multiple operator, multi-choice scenario.

Operator-Level TS (OTS) In Listing 1, each operator choice $a \in A$ has an *unknown probability of passing* θ_a , modeled as a beta distribution, and an *unknown runtime distribution*, modeled as a log-normal distribution. The domains of each distribution: beta distribution has 0-1 and log-normal has 0 – inf, make them suitable for modeling successes and runtimes [42]. To estimate these parameters before selection, the following steps are performed: We first sample the probability of passing θ_a from a Beta distribution (Line 5). We

then compute the expected number of runs n_a to pass an assertion, which is the inverse of the probability of success (Line 6). We then sample operator execution runtimes from a log-normal distribution n_a times, and compute a normalized expected runtime to pass the assertion (Line 7). Finally, the total estimated runtime for operator choice a is given in Line 8, and for each a the runtimes is stored in $\widehat{\mathcal{T}}$.

Once the estimated runtimes for all available operator choices are computed, the algorithm iteratively selects, executes, and updates operator choices until the example passes. The operator with the lowest estimated runtime is chosen as a_t^* in Line 14. The data example x_t is processed using a_t^* , and its reward (success/failure) and the runtime is observed (Line 16).

After each operator execution, the algorithm updates its internal estimates: it refines the success probability using a Beta distribution update (Line 18) and adjusts the log-normal runtime distribution parameters (μ_a, σ_a^2) based on the observed runtime y_t using the Bayesian update rule [14] in UpdateLogNormalParams (Line 20). If an operator choice fails, the corresponding action is removed from consideration, and the algorithm selects another alternative (Line 22) i.e., it incorporates rollback. This process repeats until the example successfully passes the assertion.

Algorithm 1 Single Operator-based Thompson Sampling (X, A)

```

1: # Iterate over  $N$  examples
2: for  $x_i \in X$  do
3:    $A_i \leftarrow A$  ▷ Initialize action list for current example
4:   for  $a \in A_i$  do
5:     Sample  $\theta_a \sim \text{Beta}(\alpha_a, \beta_a)$  ▷ Sample probability of passing
6:      $n_a \leftarrow \lceil 1/\theta_a \rceil$  ▷ Compute expected number of runs to pass
7:     Sample  $\tau_j \sim \text{Log-normal}(\mu_a, \sigma_a^2)$  for  $j = 1, \dots, n_a$  ▷
8:     Sample runtime for each expected run
9:      $\mathcal{T}_a \leftarrow \frac{1}{n_a} \sum_{j=1}^{n_a} \tau_j$  ▷ Compute mean sampled runtime
10:     $\widehat{\mathcal{T}}_a \leftarrow \mathcal{T}_a \cdot \frac{1}{\theta_a}$  ▷ Calculate estimated runtime
11:    Store  $\widehat{\mathcal{T}}_a$  in  $\widehat{\mathcal{T}}$ 
12:   end for
13:   while example not passed do
14:      $a_t^* \leftarrow \underset{a \in A_i}{\text{argmin}} \widehat{\mathcal{T}}$  ▷ Select best available action (fastest
15:     runtime); Subscript  $t$  indexes the processing attempt
16:     Process  $x_t$  using  $a_t^*$ ; observe pass/fail  $r_t$  and runtime  $y_t$ 
17:      $(\alpha_{a_t^*}, \beta_{a_t^*}) \leftarrow (\alpha_{a_t^*} + r_t, \beta_{a_t^*} + 1 - r_t)$  ▷ Update Beta
18:     distribution for  $a_t^*$ 
19:     UpdateLogNormalParams( $a_t^*, y_t$ )
20:      $A_i \leftarrow A_i \setminus \{a_t^*\}$  ▷ Remove chosen action from available
21:     actions
22:   end while
23: end for

```

4.2 FlexiFlow Optimizers for Multiple Operators with Multiple Choices

Thus far, we have focused on the case of a single operator, but in practice, workflows consist of multiple operators, each offering several choices.

One approach to learning in the multiple operator, multiple choice scenario is to assume operator-level independence and select the best choice for each operator individually. If the resulting configuration fails the assertion, the system proceeds to the next best configuration.

For example, in case of ID card processing, we have three operators $A = \text{Channel Picker}$, $B = \text{OCR}$ and $C = \text{Classifier}$ each with 5, 2, and 2 choices. To start, we determine the best operator choice (lowest expected runtime) independently at each of the three operators. Let's say configuration $K_1 = [A_3, B_2, C_2]$ where A_3, B_2, C_2 , represent choices within their respective operators, is the best operator choice configuration i.e., resulting in the lowest expected runtime of 10s. However, this configuration also fails the assertion. If $K_2 = [A_3, B_2, C_1]$ and $K_3 = [A_1, B_2, C_1]$ are other available configuration choices that result in 11s and 10.5s, then per OTS in Listing 1 we will choose configuration K_3 , which has the next lowest runtime.

Naively applying OTS at each operator independently has two drawbacks. First, assuming operator level independence may not be true. For example, in ID card processing, a specific channel color may be necessary for EasyOCR engine to successfully extract text but that specific channel always causes Tesseract to fail. Second, choosing each operator independently is not rollback-cost-aware, i.e. in our example, minimizing overall configuration level runtime results in choosing K_3 as the next best configuration even though the cost of incrementally running K_2 may be the lowest. Note that K_1 and K_2 share an operator prefix and the rollback cost from K_1 to K_2 maybe lower than overall cost of incurring 10.5s for running the full configuration.

We fix these two drawbacks next.

Configuration-level TS (CTS). One option is to relax operator level independence and learn at the entire configuration-level. Additionally, failed configurations perform a conditional rollback, i.e., rollback to a configuration which is estimated to have the smallest increase in runtime from the current configuration. Listing 2 describes learning at the configuration level using FlexiFlow-based TS where the set of all actions A are the set of all K configurations. For each configuration, we maintain a Beta distribution with parameters α and β and obtain the estimated runtime of a configuration $\hat{\mathcal{T}}$. Each configuration is sampled in `SelectAction` to obtain a configuration with the lowest runtime, conditional on the current best choice. The example is processed with this configuration and actual rewards and operator-level runtimes are observed. The Beta distribution parameters are updated similarly to Listing 1 Line 18, but runtime μ and σ are updated for each operator choice in the current best configuration. If the example fails to pass assertions, the next best configuration with the lowest incremental increase in runtime is chosen in `SelectAction`. This is because our objective in Equation 2 is to minimize the cost of rollbacks, $\mathcal{T}_{a_{t-1} \rightarrow a_t} / \theta_t$.

Algorithm 2 Job Configuration Selection(X, A)

```

for  $x_i \in X$  do
    # Estimate rewards and runtimes for all actions in A
     $\hat{\theta} \leftarrow \text{EstimatePassProb}(A)$ 
     $\hat{\mathcal{T}} \leftarrow \text{EstimateRuntimes}(A, \hat{\theta})$ 
    while example not passed do
        # Note:  $a_t^*$  is conditional on the rollback cost from  $a_{t-1}^*$ 
         $a_t^* \leftarrow \text{SelectAction}(\hat{\mathcal{T}}, a_{t-1}^*)$    $\triangleright a_t^*$  is the chosen action

        Process  $x_i$  using  $a_t^*$ ; observe pass/fail  $r_t$  and runtime  $y_t$ 

        UpdateRewardParams( $a^*, r_t, y_t$ )

         $A_i \leftarrow A_i \setminus \{a_t^*\}$    $\triangleright$  Remove chosen action.
    end while
end for
    
```

There are two limitations of the above learning strategy—First, the number of configurations K can be potentially large—in experiments ID processing requires learning over nearly 500 choices! Second, the configuration-level does not account for the influence of individual operators on the overall performance of a configuration. We describe a novel hybrid strategy in which we determine the best configuration based on operator level estimates.

Hybrid TS (HTS). A drawback of CTS is that we may have a very large number of configurations, making it difficult to fully explore the configuration space. We would like to use the results at the operator level to inform our selection of which configuration to try next. For example, in ID processing, let's say configuration $K_1 = [A_3, B_2, C_2]$, is the best operator choice configuration resulting in 10s of runtime but the assertion fails for this configuration. Further, let's say we have also tried a few other configurations, such as: $[A_3, B_2, C_1]$, $[A_3, B_1, C_2]$, $[A_2, B_1, C_3]$, $[A_4, B_1, C_2]$, and these configurations have also resulted in failure. Thus observing at the operator level, in all trials, operators A_3 failed thrice in total, B_2 failed twice, and C_2 failed thrice. However, we also know the entire configuration of $[A_3, B_2, C_2]$ also failed once. Even though we have only tried this configuration once, we have some idea of how it will perform based on the operators that make it up. Thus we will compute an empirical β parameter (inspired by an empirical Bayes approach [6]) for the configuration as:

$$(3 - 1) + (2 - 1) + (3 - 1)$$

i.e. we sum the failures from individual operators, but we subtract the failure of the entire configuration to avoid double counting. Similarly, we also compute an empirical α parameter for the configuration based on successes. Finally, we set a hyperparameter γ that influences how much to use the operator level information when calculating our final α and β parameters for the configuration. We “allocate” γ to the job configuration level parameters using a weighted average of successes and failures at the operator level.

Formally, let k be a job configuration in a workflow with M operators. Let α'_k and β'_k be the counts of successes and failure for the overall job configuration k . Let α_k^j and β_k^j be the counts of successes and failures for each operator j in configuration k . We first

Table 1: Comparison of different Thompson Sampling based FlexiFlow strategies. C is the total count of operator choices across operators and K is the count of configurations.

Strategy	Action Level	Operator Params	Config Params	Params Count	Rollback-Aware
Single operator with multiple choices					
OTS	Operator	α_j, β_j μ_j, σ_j^2		$O(C)$	
Multiple operators with multiple choices					
CTS	Config	μ_j, σ_j^2	α_k, β_k	$O(C + K)$	✓
HTS	Config	α_j, β_j μ_j, σ_j^2	α'_k, β'_k $\alpha_k^{em}, \beta_k^{em}$	$O(C + K)$	✓

calculate empirical Beta parameters for the configuration k , using our rule of summing the successes and failures for each operator j that makes up configuration k , but subtracting off each time that the operator was used previously as part of configuration k to avoid double counting:

$$\alpha_k^{em} = \sum_{j=1}^m (\alpha_k^j - \alpha'_k), \quad \beta_k^{em} = \sum_{j=1}^m (\beta_k^j - \beta'_k) \quad (5)$$

We then calculate α_k and β_k , which are our hybrid parameters defining the reward distribution for job configuration k under the hybrid strategy. For this, we sum the count of success and failures for job configuration k (α'_k and β'_k) with our empirical prior but normalize the empirical prior to sum to one and control its influence using the hyperparameter γ .

$$\alpha_k = \alpha'_k + \gamma \frac{\alpha_k^{em}}{\alpha_k^{em} + \beta_k^{em}}, \quad \beta_k = \beta'_k + \gamma \frac{\beta_k^{em}}{\alpha_k^{em} + \beta_k^{em}} \quad (6)$$

The hybrid strategy updates parameters after processing each data item as is done previously in other strategies. It also chooses the next best configuration similar to the conditional rollback described earlier. From a pseudocode perspective, our hybrid strategy only changes the function `UpdateRewardParams` in Listing 2 which updates the Beta distribution parameters as described in Equations 5 and 6.

Summary. We summarize the different strategies for FlexiFlow optimizers in Table 1:

- **OTS** (Operator-level Thompson Sampling) maintains probability and runtime parameters at the operator level. This approach does not work for multiple operator, multiple choice workflows.
- **CTS** (Configuration-level Thompson Sampling) maintains parameters at both operator-level (to estimate runtime) and configuration-level (to estimate probability of passing). Actions are decided at the configuration level.
- **HTS** (Hybrid Thompson Sampling) maintains probability of passing estimates both at the operator and configuration-level and uses an empirical Bayes-inspired approach to allow operator estimates to inform configuration-level selections.

OTS, HTS, and CTS are equivalent for single operator, multiple choice workflows.

5 Evaluation

Implementation. FlexiFlow implements operator choices, assertions, and online optimizers in Python. FlexiFlow is based on Scope [50] which provides a parallel dataflow language. It is written in ~20k lines of Python code. Code and experiment data presented in this paper is available [1].

A choice is programmatically expressed within an operator as follows:

```
class OCR(Processor):
    def __call__(self, input_row: Row, ...):
        img = input_row["IMG"]
        yield Row({"OCR": [lambda img=img: easy_ocr(img),
                           lambda img=img: tesseract(img)]}, ...)
```

In this implementation, the OCR operator encapsulates two possible choices—EasyOCR and Tesseract—which are represented as lambda functions. This means that either EasyOCR or Tesseract can be instantiated at any given time. Assertions are specified using functions that return `False` in case of an error, `True` otherwise. For example, the following assertion checks that the name should not be empty and should start with an upper case character:

```
def check(self, input_row: Row):
    name = input_row["name"]
    if len(name) == 0 or not name[0].isupper():
        return False, "invalid name"
    return True, None
```

While building a workflow, users can specify the optimizer. A complete workflow using operators, assertions, and optimizer is shown below. This workflow is instantiated with the HTS optimizer.

```
jb = JobBuilder(HTS(...))
n = jb.extract(ImgReader(...))
    .process(ChannelPicker(...))
    .process(OCR(...))
# ...
job = jb.union(inps=[alb, esp, ...])
    .assert(predicate=check)
    .output(CsvWriter(...))
    .build(...)
```

All observed metadata such as runtimes corresponding to operator choices and success and failures of the assertion are maintained in Redis. Redis also maintains the ancestor row lineage information. The rollback uses row-level lineage and employs fast incremental computation described in more detail in [27].

Experiment setup. We run all the experiments on a server with Intel Xeon Gold 6330 and an Nvidia A40 GPU. We evaluate whether the optimizers proposed in this paper (OTS, CTS, HTS in Table 1) can effectively lower runtimes. We perform an ablation study by selectively removing learning capabilities from the proposed optimizers. In particular, we ablate OTS to OTS-R, which does not model operator runtimes (μ_a, σ_a in Algorithm 1); it only learns the probability that each operator choice passes the assertion and assumes equal runtimes for all choices. We ablate CTS to CTS-RB, which is not rollback-aware and selects the next configuration without considering the transition cost from the previously attempted one, i.e., `SelectAction` in Algorithm 2 does not consider a_{t-1}^* .

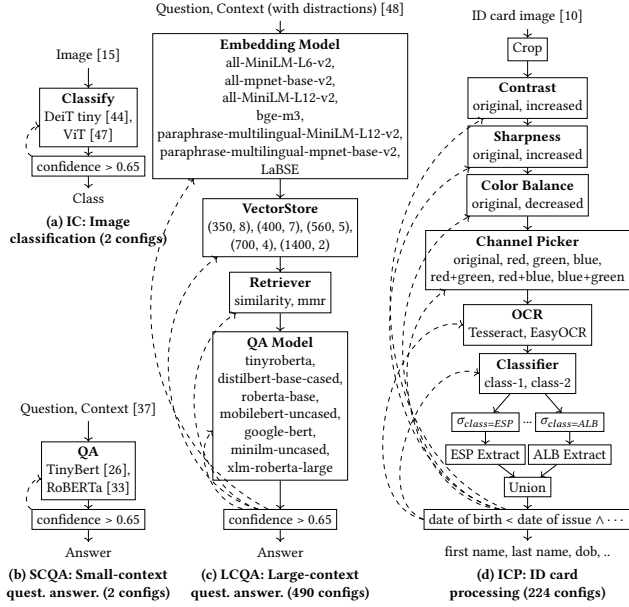


Figure 6: Real ML workflows for evaluating FlexiFlow.

ML Workflows. We evaluate FlexiFlow using four real-world ML workflows and two synthetic workflows as shown in Figure 6 and Figure 7.

Real workflows. The four real-world workflows are chosen from popular online sources described below. Each workflow includes the three core constructs—operator choices, assertions, and roll-backs—as illustrated in Figure 6. The original versions of these workflows did not include published operator choices. Instead, each configuration involving a specific model operator was published independently. The real input datasets used by the workflows are cited at the beginning of each, while the type of output produced is described at the end.

IC. This ML workflow is based on the HuggingFace image classification pipeline. The workflow processes images from the ImageNet [15] dataset. There is a single classify operator that defines two model choices: a lightweight DeiT tiny [44] model and a heavier ViT [47] model. The assertion checks if the confidence of the model prediction is above a specified threshold.

SCQA. This ML workflow is structurally similar to the previous one and is based on the HuggingFace question answering pipeline. It answers questions from the SQuAD [37] dataset containing a single QA operator that defines two model choices: a lightweight TinyBERT [26] model and a heavier RoBERTa [33] model. The assertion is similar to the one in IC. Our experiments will reveal that even though these workflows are structurally similar, FlexiFlow strategies can learn different optimal sequences of configurations which depends on runtimes and probabilities of passing assertions for different choices.

LCQA. This workflow is for multi-hop question answering and is based on the popular benchmark HotpotQA, in the "distractor"

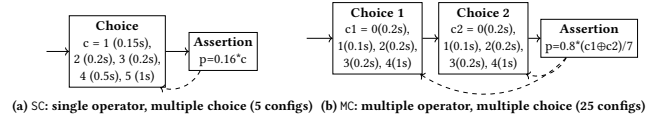


Figure 7: Synthetic workflows for evaluating FlexiFlow strategies.

mode [48]. Briefly, each question is given paragraphs from 10 different Wikipedia entries as context. Exactly two of the paragraphs contain the necessary *supporting facts* that together allow one to determine the answer.¹ The remaining paragraphs are deliberate distractions. The answers are usually one to three words long, and each predicted answer is evaluated against the ground truth using the F_1 metric.² Our workflow is similar to a standard RAG-based workflow using the LangChain orchestration framework (see, e.g., [31]). Its essential components are as follows: All the provided paragraphs are first concatenated and then split into chunks. An embedding vector is generated for each chunk using one of 7 embedding models (the first operator; see Figure 6(c) for model choices). The size of each chunk is chosen from 5 different options (second operator; called type of VectorStore). The embeddings of the question and the chunks are then used to retrieve the most relevant chunks, again using one of two retrieval mechanisms (third operator). The number of chunks retrieved can also vary, but for implementation reasons, this is a fixed number for each VectorStore option. So, e.g., the VectorStore option (350, 8) is the option in which the size of each chunk is 350 characters and 8 of the most relevant chunks are returned by the retriever. The relevant chunks are fed into one of 7 question-answering (QA) models (fourth and last operator).³ Finally, our assertion is that the confidence score computed by the QA model should meet a minimum threshold, which was set to 0.65 in our experiments. This workflow has a total of 490 configurations.

ICP. This ML workflow is inspired from articles describing KYC pipelines [9] deployed in the industry and open KYC Github repositories [30, 45]. This workflow extracts fields from ID card scans from the MIDV-2020 dataset [10]. It crops the images to make them upright and then does various image cleaning operations like changing contrast, sharpness, color balance, and picks color channels to make the images more amenable to OCR. Then, it uses OCR engines and country-specific field extractor models to extract fields like first name, last name, and date of birth. Extracted fields are evaluated against ground truth using the F_1 metric.

Image cleaning operations are made optional by keeping the original image as one of the operator choices. The OCR operator specifies two different OCR engines as choices. The classifier operator specifies top-2 predictions from a ResNet [21] classifier as choices. The assertions verify several data integrity constraints, such as ensuring that the extracted name is not null and that the extracted date of birth precedes the extracted date of issue. This workflow has a total of 224 configurations.

¹Having to analyze facts across two separate paragraphs is what makes this task "multi-hop."

²Precision is defined as the ratio of the number of tokens common to both the prediction and the ground truth to the number of tokens in the prediction. Recall and F_1 are defined similarly. See, e.g., [46].

³Our models are general purpose QA models commonly used in HuggingFace/LangChain and not specially fine-tuned to identify supporting facts in the HotPotQA dataset.

Synthetic workflows. Since we do not know the optimal sequence of configurations for real-world ML workflows, we also evaluate FlexiFlow strategies with synthetic workflows, as illustrated in Figure 7, so that we can compare FlexiFlow performance to an optimal strategy. We calculate each workflow’s optimal sequence of configurations, based on Equation 2, to evaluate whether and how quickly FlexiFlow strategies learn to provide optimal runtimes. Each synthetic workflow contains one or two choice operators, a single assertion, and the ability to rollback to any choice operator.

The pass probability of the assertion depends on the selected choice/(s). In SC, the probability of passing the assertion is kept as $0.16c$, where c is the choice index. Therefore, even the best choice has a probability of 0.8 *i.e.*, it does not always pass the assertion. This mimics the behavior of models in real world ML workflows.

Each choice is associated with a runtime cost. For instance, choice $c = 1$ in SC incurs a runtime of 0.15 seconds. We chose these values to mimic real world situations where heavier models are often more accurate but also take more time. For example, choice 5 has a higher probability (0.80) of passing the assertion than choice 4 (0.64), but choice 5 takes double the time ($1s$) to run than the choice 4 ($0.5s$).

For MC, the assertion is based on the combination of choice index values. We again keep the maximum probability as 0.8 so that none of the configurations always pass the assertion. To stress test the learning task of FlexiFlow strategies, we keep the choices in the two operators $c1$ and $c2$ dependent on each other by setting the probability $0.8 \times (c1 \oplus c2)/7$, where $\oplus$ denotes the bitwise XOR operation. For example, $c1 = 4$ has a 0.8 probability of passing assertions when $c2 = 3$ but has a 0 probability with $c2 = 4$.

As shown in the figure, each choice operator provides 5 possible choices. While the number of configurations can be varied, we kept a maximum of 25 configurations so that the optimal sequence of configurations could be computed by a brute-force approach with sequence pruning. We omit the details of this brute-force approach due to space limitations. Prior knowledge of optimal ordering allows us to evaluate whether and how quickly FlexiFlow strategies converge to the same runtime as the optimal strategy. We also ran our experiments with other runtimes and pass probabilities in the synthetic workflows. Since the overall results were not much different, we skip them here due to space constraints.

Evaluation Goals and Baselines. We evaluate FlexiFlow on three fronts:

- (1) Do FlexiFlow’s learning strategies quickly learn optimal runtimes for diverse ML workflows? This evaluation helps to assess the learning rate of the single operator and multi operator learning strategies. We evaluate this question using both synthetic and real workflows. In synthetic workflows, the optimal sequence serves as the baseline for comparison. In real workflows, since the optimal sequence is not available, we assess how quickly does a strategy converge to stable regret in the presence of a large number of configurations *i.e.*, determine the scalability of strategies. We also present case studies to show how optimizers learn efficiently even in workflows with hundreds of configurations. (Section 5.1)
- (2) Do FlexiFlow’s programming abstractions—operator choices and assertions—improve accuracy compared to the single best

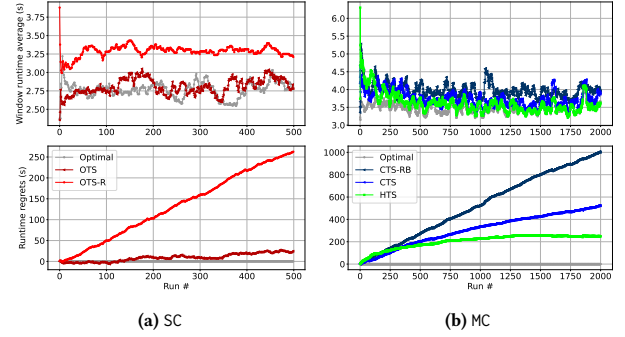


Figure 8: Modeling runtime and being rollback aware is important for strategies to learn.

configuration? We evaluate this on real ML workflows by separately running each workflow with individual configurations and comparing each configuration’s accuracy with the accuracy provided by FlexiFlow. (Section 5.2)

- (3) Finally, how does FlexiFlow’s performance compare with other similar dataflow systems? For this goal, we use real workflows and compare their performance on FlexiFlow and state-of-the-art dataflow systems Spark [49] and Flink [11]. (Section 5.3)

5.1 Performance of FlexiFlow optimizers

In this section, we evaluate whether the optimizers proposed in this paper (OTS, CTS, HTS in Table 1) can lower runtimes for both synthetic and real-world ML workflows. We perform ablation studies by also evaluating OTS-R (OTS without runtime modeling) and CTS-RB (CTS without rollback awareness).

5.1.1 Performance of optimizers on synthetic workflows. For each synthetic workflow, we first run it multiple times with its optimal sequence of configurations to get a series of optimal runtimes. Due to probabilistic assertions, choices may fail and cause rollbacks. Hence, the runtimes vary across different runs. Next, with a given optimizer, we run the workflow multiple times, with the optimizer updating itself after each run, to get a series of optimizer runtimes. We then compare the series of optimizer runtimes against that of optimal runtimes. Since runtimes vary across each run, we show runtime trends by plotting a sliding-window runtime average. In addition, we plot the cumulative regret curves for each optimizer. Cumulative regret is measured as the cumulative difference between the achieved runtime and the optimal runtime. Plotted as a series, cumulative regret curves show how fast an optimizer approaches the optimal runtime; a good learning strategy will show a “log-like shape” in its cumulative regret curve with a rapidly decreasing slope. Note that because of probabilistic assertions, learning strategies can sometimes get lucky and finish faster than the optimal strategy, leading to negative regret.

SC. We evaluate OTS and OTS-R for SC. We see from Figure 8a that the runtime-agnostic OTS-R is unable to reach the optimal runtime. This is because OTS-R simply maximizes probability of passing assertions. It learns to run the sequence $c = [5, 4, 3, 2, 1]$, but the optimal sequence for SC is $c = [3, 2, 4, 1, 5]$ due to different runtimes of choices. The runtime-aware multi-armed bandit OTS

is able to learn the optimal order of choices maintaining a much lower regret than OTS-R.

MC. We next evaluate CTS, HTS, and CTS-RB for MC. From Figure 8b, we observe that the CTS-RB, that does not take rollback order into account, cannot learn the optimal order. This is because this learning strategy just picks configurations with the lowest expected runtime without considering the computational structure of the workflow. For example, in the last run, CTS-RB tries $(c_1 = 3, c_2 = 4)$ followed by $(c_1 = 4, c_2 = 2)$. In the same run, both CTS and HTS try $(c_1 = 4, c_2 = 1)$ followed by $(c_1 = 4, c_2 = 0)$. Switching from $(c_1 = 4, c_2 = 1)$ to $(c_1 = 4, c_2 = 0)$ avoids re-executing $c_1 = 4$. This shows that rollback awareness is essential for learning in workflows with multiple choice operators.

Further, we observe in Figure 8b that HTS learns much faster than CTS, resulting in lower cumulative regret. This implies in HTS augmenting the probability estimations for configurations with the estimations for individual choices makes learning more effective. *Our key observations are as follows: (i) accurately modeling both runtime and the probability of assertion success is crucial for determining the optimal execution order; (ii) while the OTS strategy is effective for workflows involving a single choice operator, workflows with multiple choice operators require strategies that are aware of potential rollbacks; and (iii) the hybrid strategy HTS—combining configuration-level learning with operator-level insights—achieves faster convergence compared to the configuration-only CTS approach.*

5.1.2 Performance of optimizers on real-world ML workflows. In this section, we shift our focus to real-world workflows and analyze how these strategies impact runtime performance. Specifically, we determine the effect of OTS on on runtimes of single operator workflows, IC and SCQA, and the impact of HTS on runtimes of multiple operator workflows, LCQA and ICP.

Each workflow is executed with 1 input per run. After each run, we unload all the ML models from the GPU memory and stop all workers. Since, the optimal sequence of configurations for these real workflows is unknown, we evaluate how quickly learning strategies converge by reporting their cumulative runtime average. We also analyze the final configuration sequence learned by optimizers after convergence.

OTS Convergence Behavior on Real Workflows. OTS learns effectively within the first 50 runs and consistently maintains low runtime for both IC and SCQA (figures omitted due to space constraints). For IC, OTS learns to favor the lightweight DeiT-Tiny model over the heavier ViT model. DeiT-Tiny loads and runs inference in 2.0 seconds, compared to 2.4 seconds for ViT, and satisfies the assertion in 82% of cases—making it a strong first choice. However, for SCQA, OTS learns to prioritize the heavier RoBERTa model over the lighter TinyBERT. The optimizer observes that the runtime difference is minimal—1.4s for TinyBERT versus 1.5s for RoBERTa. Moreover, the lightweight model often fails to satisfy the assertion, making the heavier model the better choice.

HTS Convergence Behavior on Real Workflows. Figure 9a shows that the HTS runtime for the LCQA workflow converges after about 1000 runs. In its final run, the optimizer selects the configuration

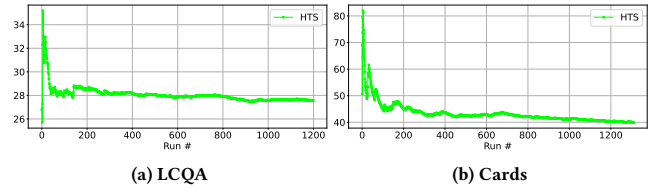


Figure 9: FlexiFlow optimizers can effectively improve run-times for real-world ML workflows.

<all-MiniLM-L6-v2, (700, 4), mmr, google-bert>⁴. Interestingly, this choice is not the one with the highest scores—its $\alpha = 36.4$ ⁵ and the empirical mean of 0.46, rank only third among the 490 configurations. However, because HTS is rollback aware, it explores other potential configurations probabilistically.

Thus, after this initial configuration, the optimizer systematically explores the configurations as follows: (i) it tests 6 different QA models; (ii) it changes the retriever from mmr to similarity in the next 7 configurations; (iii) it runs with the chunk size 400 for the next 14 configurations, and, finally (iv) it changes the embedding model to LaBSE.

HTS Converges Similarly but Learning Behavior Differs in the ICP Workflow. Figure 9b shows HTS also improves runtimes for the ICP workflow, though the learned configuration sequence behaves quite differently from LCQA. In LCQA, multiple operators contribute non-trivially to the total runtime. But in ICP, only the OCR operator has a substantial runtime; all the other choices take less than 80ms. Further, it learns that Tesseract passes 9 times less often than EasyOCR. Consequently, instead of following a bottom-up strategy as in LCQA, HTS adapts to the runtime profile of ICP as follows: It delays trying Tesseract until the 17th configuration in the final sequence; The first configuration disables most image-cleaning steps; When that fails, it tries the next prediction from the classifier since it does not need another model inference; Next, it tries modifying sharpness and later switches to a red+green color channel.

Overall, we find that FlexiFlow optimizers learn efficiently even in workflows with hundreds of configurations. By leveraging assertion success rates, operator runtimes, and workflow structure, the optimizers adapt their strategies—favoring lightweight models when feasible and selecting more reliable ones when necessary.

5.2 Accuracy of FlexiFlow

We now examine how and why FlexiFlow delivers more accurate results by considering the question answering workflow LCQA and the document processing workflow ICP. We compare the accuracy of FlexiFlow with that of individual workflow configurations run separately on identical input. Instead of individually running the large number of configurations, we run a reduced version of the workflows. For LCQA, we use two of the seven embedding models: all-MiniLM-L6-v2 and LaBSE, and two of the seven QA models: xlm-roberta-large and tinyroberta, totaling 40 configurations. For ICP, we remove contrast, sharpness, and color balance image cleaning

⁴Recall a configuration for LCQA specifies operator choices for (Embedding Model, VectorStore chunk size and number of chunks retrieved, Retriever, QA Model); also see Figure 6 (c).

⁵Recall that HTS sums up a fraction of α and β values from the operator level. They may therefore not be an integer.

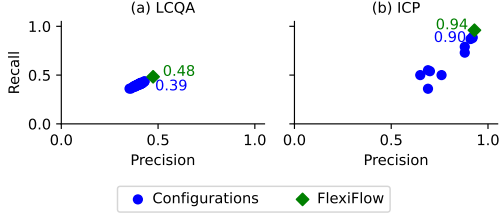


Figure 10: Effectiveness of using operator choices in ML workflows. FlexiFlow achieves a higher F_1 score than the best configuration.



Figure 11: Failure modes of LCQA configurations are uncorrelated.

operators, and remove red+blue and red+green options from ChannelPicker, totaling 20 configurations (refer Figure 6(d)). FlexiFlow version also runs these reduced workflows.

FlexiFlow achieves an F_1 score of 0.48 (see Figure 10 (a)), a relative improvement of 23% over the best single underlying configuration—(all-MiniLM-L6-v2, (350, 8), similarity, xlm-roberta-large), which only achieves a score of 0.39.⁶ As the figure shows, FlexiFlow is better in both precision and recall.

The reason for the significant improvement in performance is that different underlying configurations are *uncorrelated*—they differ in the examples they get wrong. For instance, Figure 11 shows that the configuration with the best F_1 metric gets 174 questions correct (using the *exact match* metric). Among the remaining questions, a different configuration (LaBSE, (560,5), similarity, tinyroberta) gets 169 questions correct.⁷

Further, the estimated confidence score is a reliable indicator of the likelihood of the configuration being correct, which is exploited by FlexiFlow to switch to a different configuration when needed. To illustrate, here is a question and the corresponding answers provided by 4 of the configurations, in increasing order of confidence. The incorrect answers have been crossed out.

Q: “Human Error” is the season finale of the third season of a TV show that aired on what network?

- A₁: ~~USA Network~~. (all-MiniLM-L6-v2, (1400, 2), mmr, xlm-roberta-large) with confidence 0.00.
- A₂: ~~House~~. (all-MiniLM-L6-v2, (350, 8), mmr, xlm-roberta-large) with confidence 0.05.
- A₃: ~~ABC~~. (all-MiniLM-L6-v2, (350, 8), similarity, xlm-roberta-large) with confidence 0.30.
- A₄: Fox. (LaBSE, (700, 4), mmr, xlm-roberta-large) with confidence 0.76.

The configuration corresponding to A₃ is actually the single best configuration overall (recall that it achieves an F_1 of 0.39). Yet, here it has low confidence in its answer, which also happens to be

⁶We chose general purpose QA Models rather than ones specially fine-tuned for HotpotQA because our primary objective here is to evaluate the performance of FlexiFlow w.r.t. a given a set of configurations. As such, our models score lower than the state-of-the-art for this dataset.

⁷Observe that the former configuration had the best F_1 score, while the latter configuration had a higher exact match score.

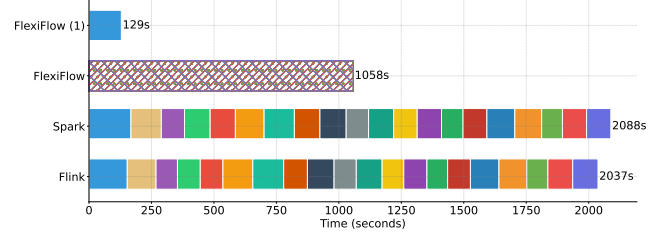


Figure 12: Comparing runtime of FlexiFlow with simulating rollback in other dataflow systems.

incorrect. Further, it is not the QA model that needs to be changed, but the embedding model and the VectorStore (use larger chunks)—see configuration corresponding to A₄.

Contrary to intuition, it is not always the larger models that provide the correct answer or have the most confidence. For the question “The Oberoi family is part of a hotel company that has a head office in what city?” the heavier model in (all-MiniLM-L6-v2, (350, 8), similarity, xlm-roberta-large) does extract the correct answer (“Delhi”) but with a confidence of only 0.01. So FlexiFlow turns to the lighter model in (all-MiniLM-L6-v2, (350, 8), similarity, tinyroberta), which gives the same answer but with a confidence of 0.98. Similarly for ICP, Figure 10(b) shows that the different ICP configurations can reach a maximum F_1 metric of 0.90, but when using operator choices with FlexiFlow, the F_1 metric reaches 0.94. The reason for this improvement is again uncorrelated failures between the configurations. We skip more qualitative analysis due to space constraints.

Overall, we conclude that using operator choices using FlexiFlow could improve performance by up to 23%.

5.3 Runtime Performance of FlexiFlow

Since existing systems lack native support for operator choices, assertions, and rollbacks, the accuracy advantage shown above could be achieved only by running the workflow configurations sequentially. We create this setup by writing Spark and Flink jobs for each of the 20 configurations in the reduced ICP workflow described above. All the cards are input to the workflow configuration with the highest F_1 score. The outputs are checked with the same assertion as in Figure 6(d). Cards that fail the assertion are input to the next configuration. This iterative process is continued until all the configurations are exhausted, effectively simulating rollback. FlexiFlow was evaluated using the same input cards to compare the total processing time. Each system is limited to 10 CPU cores. Each experiment was repeated 5 times. The variation in runtimes was negligible.

Figure 12 compares the total runtime for FlexiFlow with sequential execution of workflow configurations in Spark and Flink. Runtime of each execution in the sequence is shown in a different color for Spark and Flink. The figure shows that FlexiFlow completes the run 48% faster than sequential executions in Flink and 49% faster than those in Spark. The reason for this improvement is that FlexiFlow can reuse intermediate results during rollbacks and can better leverage data parallelism as it can run different configurations for different inputs simultaneously.

We also compare the general performance of FlexiFlow with Spark and Flink by running just the best ICP configuration while

disabling all the operator choices and assertions from FlexiFlow on identical inputs and setup. Figure 12 shows this as FlexiFlow (1). FlexiFlow can finish this run 15% faster than Flink and 22% faster than Spark⁸.

Overall, we conclude that the general performance of FlexiFlow is comparable to state-of-the-art systems. The native support for operator choices, assertions, and rollbacks in FlexiFlow can improve runtimes by 48% compared to sequentially executing workflow configurations.

6 Discussion

FlexiFlow reduces programmer burden by letting them specify operator choices instead of maintaining multiple workflow instances. FlexiFlow allows a diversity of choices. FlexiFlow reduces management burden of prioritizing between multiple workflow instances using bandit-based strategies.

This paper only takes the first step in letting programmers rely on FlexiFlow strategies to pick the right choice at each operator. In future work, the proposed strategies can also take input data characteristics into account to potentially learn different optimal sequences of configurations for data with different characteristics (e.g., cards from different countries in ICP). FlexiFlow is most effective when failure modes of configurations are uncorrelated. But, failures may not be completely uncorrelated. For example, when a heavy model fails to predict correctly, it might be more likely that the light model will also fail. FlexiFlow strategies do not separately learn conditional probability of passing assertions which may be crucial for such cases. Modeling conditional probabilities without managing large amounts of state is also part of our future work. FlexiFlow strategies currently focus on lowering runtimes. But many batch workflows are more dollar-cost sensitive than they are runtime sensitive, such as call commercial LLMs less often. We believe a similar problem formulation may suffice for reducing dollar costs.

7 Related Work

Workflow and DataFlow Systems. FlexiFlow is a dataflow system where the operator order within DAG defines the data dependencies. The operators follow pipeline execution in that operators execute as soon as their input data is available. This is similar to Dataflow systems such as MapReduce [13] and others [25, 50]. However these systems do not have support for alternative operator choices, assertions, and rollbacks to improve on overall workflow accuracy. Timely dataflow [35] allows for some form of rollback, but without alternative operator choices.

The rollback and incremental computation in FlexiFlow is similar to incremental updates in incremental view maintenance [2, 7, 36, 51], streaming computation [3, 8], and provenance-based selective replay [23]. These systems typically handle updates at the input, such as base table updates in incremental view maintenance or stream appends in streaming computation. In contrast, FlexiFlow allows updates at intermediate outputs during rollbacks.

Programming Models. Operator choices are similar to McCarthy’s ambiguous functions (Amb) [34] where the language runtime tries to pick an alternative to satisfy assertions. Operator choices are

also similar to choice blocks in template programs [4, 41] where programmers just write a template and let an SMT solver choose the right block so that the overall program meets a specification. To the best of our knowledge, FlexiFlow is the first to apply the concept of operator choices within the context of a dataflow system. The target use case of FlexiFlow is different from previous work.

DataFlow Debugging. Systems such as Amber [29] and those based on Spark including Titian [24], BigSift [20], BigDebug [19], and TagSniff [12] support debugging primitives such as breakpoints (to pause and resume execution) and watchpoints (to inspect intermediate data). While these can be used for error detection, often they require manual changes for error correction. Similarly, model assertions perform runtime checks to detect model errors at runtime [28]. However, model assertions work in the context of a single model whereas we detect errors on overall workflows and correct them by rolling back to alternate workflow configurations.

Online Learning. FlexiFlow strategies build upon Thompson Sampling [40, 42, 43], which samples all actions that could plausibly be optimal, while shifting sampling away from those that are unlikely to be optimal. Theoretically, TS has been shown to achieve asymptotically optimal logarithmic regret in stochastic MAB settings [40]. While a formal regret analysis for our approach is beyond the scope of the current paper, experimentally we benefit from learning the right choices. We also note that, while there are situations that do not require exploitation or exploration, or situations where it is better to exploit a high performing suboptimal action than to invest resources exploring actions, often modeling such situations depends upon availability of some high-quality historical data. We currently do not assume access to any such historical data and we learn in an online setting.

Confidence Estimation. When the assertion in FlexiFlow is a confidence threshold (as in IC, SCQA, LCQA), our system is related to the work done in confidence estimation which is applied to, e.g., out-of-distribution detection and selective prediction. Examples of such work include Bayesian neural networks, ensemble of neural networks, and temperature scaling (see, e.g., the surveys [17, 18]). Unlike our work, these do not attempt to mitigate the runtime cost of the ensemble by learning an optimal sequence of models to try.

8 Conclusion

We showed that in many ML pipelines, relying on a single model to perform inference across all data batches often fails to maximize accuracy and thus overall performance. Our FlexiFlow system, incorporates operator choices, assertions, and rollbacks and uses a novel multi-armed bandit approach to perform automatic model switching and thus improves overall accuracy and efficiency of various ML pipelines such as document processing, image classification, and question-answering tasks.

Moving forward, our approach opens new avenues for AutoML-based systems that autonomously adapt to evolving data distributions. Future work will explore incorporating intermediate data characteristics, such as the country of an ID card, into the learning strategies. Ultimately, FlexiFlow represents a step toward more intelligent, data-aware workflow systems that reduce manual intervention and enhance the reliability of ML-driven automation.

⁸This performance difference might be because our implementations are written in Python. This may cause Python-JVM interaction overheads for Spark and Flink.

References

- [1] [n.d.]. Source code and experiments. <https://anonymous.4open.science/r/flexiflow-5BC6/>.
- [2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (jun 2012), 968–979. <https://doi.org/10.14778/2336664.2336670>
- [3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1792–1803. <https://doi.org/10.14778/2824032.2824076>
- [4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. <https://doi.org/10.1109/FMCAD.2013.6679385>
- [5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, Melbourne, Australia, 1383–1394. <https://doi.org/10.1145/2723372.2724797>
- [6] James O. Berger and James O. Berger. 1985. *Statistical decision theory and Bayesian analysis*. Springer-Verlag, New York. http://www.amazon.com/Statistical-Decision-Bayesian-Analysis-Statistics/dp/0387960988/ref=sr_1_11?ie=UTF8&qid=1403880466&sr=8-11&keywords=Bayesian+statistics
- [7] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently Updating Materialized Views. *SIGMOD Rec.* 15, 2 (jun 1986), 61–71. <https://doi.org/10.1145/16856.16861>
- [8] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (March 2023), 1601–1614. <https://doi.org/10.14778/3587136.3587137>
- [9] Duy Tuan Bui. 2021. Applications of Machine Learning in eKYC’s identity document recognition. (2021).
- [10] Konstantin Bulatov, Ekaterina Emelianova, Daniil Tropin, Natalya Skoryukina, Yulia Chernyshova, Alexander Sheshkus, Sergey Usilin, Zuheng Ming, Jean-Christophe Burie, Muzzamil Luqman, and Vladimir Arlazarov. 2021. MIDV-2020: A Comprehensive Benchmark Dataset for Identity Document Analysis. <https://doi.org/10.48550/arXiv.2107.00396>
- [11] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.
- [12] Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, Zoi Kaoudi, and Saravanan Thirumuruganathan. 2019. Tagsniff: Simplified big data debugging for dataflow jobs. In *Proceedings of the ACM Symposium on Cloud Computing*. 453–464.
- [13] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters (OSDI’04). USENIX Association, 10.
- [14] Morris H. DeGroot and Mark J. Schervish. 2011. *Probability and Statistics* (4th ed.). Pearson Education, Upper Saddle River, NJ.
- [15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 248–255.
- [16] Jing Gao, Bolin Ding, Wei Fan, Jiawei Han, and S Yu Philip. 2008. Classifying data streams with skewed class distributions and concept drifts. *IEEE Internet Computing* 12, 6 (2008), 37–49.
- [17] Jakub Gawlikowski, Christian R. N. Tassi, Muhammad Ali, et al. 2023. A survey of uncertainty in deep neural networks. *Artificial Intelligence Review* 56, Suppl 1 (2023), 1513–1589. <https://doi.org/10.1007/s10462-023-10562-9>
- [18] Jiahui Geng, Fengyu Cai, Yuxia Wang, Heinz Koepl, Preslav Nakov, and Iryna Gurevych. 2023. A Survey of Confidence Estimation and Calibration in Large Language Models. arXiv preprint arXiv:2311.08298v2. <https://arxiv.org/abs/2311.08298> Last revised March 25, 2024.
- [19] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*. 784–795.
- [20] Muhammad Ali Gulzar, Siman Wang, and Miryung Kim. 2018. Bigsift: automated debugging of big data analytics in data-intensive scalable computing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 863–866.
- [21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR abs/1512.03385* (2015). arXiv:1512.03385 <http://arxiv.org/abs/1512.03385>
- [22] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.
- [23] Robert Ikeda, Semih Salihoglu, and Jennifer Widom. 2011. Provenance-Based Refresh in Data-Oriented Workflows. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) (CIKM ’11). Association for Computing Machinery, 1659–1668. <https://doi.org/10.1145/2063576.2063816>
- [24] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (nov 2015), 216–227. <https://doi.org/10.14778/2850583.2850595>
- [25] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.
- [26] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2019. TinyBERT: Distilling BERT for Natural Language Understanding. *CoRR abs/1909.10351* (2019). arXiv:1909.10351 <http://arxiv.org/abs/1909.10351>
- [27] Abhilash Jindal, Kaustubh Beedkar, Vishal Singh, J Nausheen Mohammed, Tushar Singla, Aman Gupta, and Keerti Choudhary. 2024. Reactive Dataflow for Inflight Error Handling in ML Workflows. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning*. 51–61.
- [28] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. 2020. Model assertions for monitoring and improving ML models. *Proceedings of Machine Learning and Systems* 2 (2020), 481–496.
- [29] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. 2020. Amber: A Debuggable Dataflow System Based on the Actor Model. *Proc. VLDB Endow.* 13, 5 (jan 2020), 740–753. <https://doi.org/10.14778/3377369.3377381>
- [30] Suman Kunwar. [n.d.]. Github: OCR ID card parser. <https://github.com/sumn2u/ocr-id-card-parser>. Accessed: 2025-06-27.
- [31] LangChain Development Team. 2025. *Build a Retrieval Augmented Generation (RAG) App: Part 1*. LangChain. <https://python.langchain.com/docs/tutorials/rag/> Accessed: 2025-06-27.
- [32] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer, 21–37. https://doi.org/10.1007/978-3-319-46448-0_2
- [33] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR abs/1907.11692* (2019). arXiv:1907.11692 <http://arxiv.org/abs/1907.11692>
- [34] John McCarthy. 1959. A basis for a mathematical theory of computation. In *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 33–70.
- [35] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 13)*. 439–455.
- [36] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 511–526. <https://doi.org/10.1145/2882903.2915246>
- [37] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don’t Know: Unanswerable Questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, Melbourne, Australia, 784–789. <https://doi.org/10.18653/v1/P18-2124>
- [38] Joseph Redmon, 绍维克Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.
- [39] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in Neural Information Processing Systems*, Vol. 28.
- [40] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2020. A Tutorial on Thompson Sampling. arXiv:1707.02038 [cs.LG] <https://arxiv.org/abs/1707.02038>
- [41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. Association for Computing Machinery, 404–415. <https://doi.org/10.1145/1168857.1168907>
- [42] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. <http://incompleteideas.net/book/the-book-2nd.html>
- [43] William R. Thompson. 1933. On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika* 25, 3/4 (1933), 285–294. <https://doi.org/10.2307/2332286>

- [44] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. 2021. Training data-efficient image transformers and distillation through attention. In *International Conference on Machine Learning*, Vol. 139. 10347–10357.
- [45] Muhammad Usman. [n.d.]. Github: ID card OCR. <https://github.com/MuhamaadUsman/ID-Card-OCR>. Accessed: 2025-06-27.
- [46] Wikipedia contributors. 2025. Precision and Recall. https://en.wikipedia.org/wiki/Precision_and_recall. Accessed: 2025-06-27.
- [47] Bichen Wu, Chenfeng Xu, Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Zhicheng Yan, Masayoshi Tomizuka, Joseph Gonzalez, Kurt Keutzer, and Peter Vajda. 2020. Visual Transformers: Token-based Image Representation and Processing for Computer Vision. arXiv:2006.03677 [cs.CV]
- [48] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- [49] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*. 15–28.
- [50] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *The VLDB Journal* 21, 5 (Oct 2012), 611–636. <https://doi.org/10.1007/s00778-012-0280-z>
- [51] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. 1995. View Maintenance in a Warehousing Environment. *SIGMOD Rec.* 24, 2 (may 1995), 316–327. <https://doi.org/10.1145/568271.223848>