Automating ML Model Development at Scale

Kaiyuan Wang
Google Inc
kaiyuanw@google.com

Yang Li
Google Inc
yangliyl@google.com

Junyang Shen
Google Inc
junyangs@google.com

Kaikai Sheng
Google Inc
ksheng@google.com

Yiwei You
Google Inc
davidyou@google.com

Jiaqi Zhang
Google Inc
jiaqizh@google.com

Srikar Ayyalasomayajula
Google Inc
srikayy@google.com

Julian Grady
Google Inc
jpg@google.com

Martin Wicke
Google Inc
wicke@google.com

***Abstract*—Google has a large team of machine learning (ML) developers working on a large number of ML models. ML model development suffers from long edit/validate cycles compared to traditional software development. This makes it tedious and time consuming for modeling teams to propagate ML innovations to their models. We present HEINZELMAENNCHEN, an ML modeling automation system, which allows users to apply semantically specified modeling changes to models and evaluate them at scale.**

Three insights are key to creating this system: Automatic code modification allows us to mechanically apply modeling changes to a wide variety of models. Workflow automation systems are well suited to operate complex ML training machinery as if they were humans, saving significant manual effort. And finally, given a large enough model population, even imperfect automatic modeling with a lower-than-human success rate will generate significant aggregate gains.

In this paper, we describe the design and an implementation of our system. We also evaluate the system performance and include an empirical study to demonstrate the utility and critical impact of the system. Our system is widely used by hundreds of ML developers and it significantly accelerates model development on hundreds of production models.

***Index Terms*—SE for AI, ML Model Development**

I. INTRODUCTION

Machine learning (ML) has become an essential part of many companies' business, occupying significant numbers of engineers improving ML models. Traditional software engineering allows engineers to validate their changes quickly, by syntax checking, compilation, and unit testing. On the other hand, ML engineering [1] suffers from long edit/validate cycles since the effects from modeling changes can often only be understood once a model has been retrained and evaluated, a process that can take days or even weeks, depending on the size and complexity of the model. After offline evaluation shows promise, live A/B experiments are needed to determine with certainty whether a change has a positive impact on the business. To find a promising change that can actually be launched, developers often need to create and evaluate hundreds of candidate changes.

This makes automation an obvious productivity booster. The traditional approach in such situations is to create integrated end-to-end ML platforms [2], [3], [4], [5]. These aim to streamline the development process by standardizing model development and reducing the amount of time spent on making model changes and shepherding those changes through various pieces of infrastructure. However, developers still need to write and manually edit their models as code in a model-building framework such as TensorFlow [6] or PyTorch [7].

Another avenue for reducing the need for manual experimentation is to apply black-box [8] or machine learning based optimization [9] techniques to assist engineers in the process of identifying the most promising changes. Models are code, and truly automating code modification is difficult in general. Therefore, even with those techniques, it is necessary to make manual changes to each model separately.

Approaches to fully automatically construct models from data exist [10], [11], but such approaches can only create models from scratch. At Google, we must operate on models that have been carefully tuned, often for years. Automatically created models cannot compete in this environment.

In this paper, we propose HEINZELMAENNCHEN, a system that improves existing models, creating changes that are then evaluated for quality and efficiency. To the best of our knowledge, our system is the first to automate the experimentation user journey to this extent.

Automatically and reliably editing code is difficult outside narrowly defined domains. Fortunately, ML models are often written in a declarative form: A program creates a background representation (e.g., a Keras `Model` class or a TensorFlow `GraphDef`) which is executed later, possibly on a different machine/process. This allows us to work not only on the code representation of the model, but on the background representation, which is a significantly easier problem.

Still, model changes present a task too difficult for fully automated, direct optimization. Instead, we allow users to encode changes that they have developed for a model in an abstract form, a *recipe*. Concretely, a recipe is a program that makes model code edits to modify the semantic of a model. Such recipes are applicable to a wide variety of models, and once encoded, can be applied automatically at scale without human involvement.

To achieve full automation, we encode the end-to-end modeling and evaluation workflow into *protocols*, which actuate different systems as a human user would. In combination, recipes and protocols allow us to automate the full end-to-end experimentation user journey. Our users can select the type of recipe and protocol to execute, and are presented (after the required training time) with evaluation results.

This paper makes the following contributions:

2832-7659/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE-SEIP66354.2025.00020

- We propose mechanisms that allow ML engineers to encode a modeling change as a *recipe*, an abstracted and generalizable representation of a modeling change.
- We describe a scalable and reliable end-to-end automation system of offline experimentation. This means that modeling changes are not only automatically created, our system starts and monitors the training process, and performs configurable evaluations after training is complete.
- We demonstrate the effect of our system in Ads at Google, with hundreds of developers working on hundreds of models, creating $O(10\text{k})$ new experimental model variants per year. We observe increases in productivity, faster spread of novel and complex techniques through the model population, as well as increased adoption of simpler techniques. The system also accelerates revenue critical model launches.

II. Background

Our system operates in a complex environment of existing tools, optimized for scale and efficiency. Below we describe the main features of these adjacent technologies, as they influence the design choices in HEINZELMAENNCHEN.

A. Training Service

The Ads training service at Google is a machine learning system for continuous training of large-scale deep neural networks (DNNs) on non-stationary data. Training examples are visited in chronological order and the model is continuously trained as new data arrives.

The training service supports a very large population of active models (around O(1000) concurrently); model management operations are accessible through a command line interface and a Web UI. While these interfaces were designed for use by humans, they are standardized enough for operation by an automation service. The training service manages model training for the most revenue-critical models at Google, and therefore provides strong guarantees on repeatability and execution reliability. Nevertheless, given the scale and number of models in training, it regularly surfaces error to users for various reasons, and our automation system has to handle a variety of errors to achieve a high training success rate.

Each model in the training service has a model construction plan and a training job configuration. The model construction plan describes the training data and the model hyperparameters such as learning rate and batch size, as well as the model architecture and how to process input features. It is implemented in Python and conceptually very similar to a Keras [12] *Model*. The plan encodes the model semantics using abstractions such as layers and their connections. The information is used to construct a TensorFlow graph, which is then executed on dedicated workers during training. The training job configuration controls the behavior of the training jobs, which includes the model priority, the number of Tensor Processing Units (TPUs) to use, etc.

B. Orchestration Service

We use an orchestration service that can execute arbitrary steps and is configured using an acyclic dependency graph. A step is a binary that could implement arbitrary logic. The system is functionally equivalent to publicly available orchestration systems such as Apache AirFlow [13].

This orchestration service provides a hosted solution with strong execution guarantees and primitives for error handling. This is essential to handle user-facing errors generated by the training service (and other components).

C. Neural Architecture Search

To find a model architecture with optimal accuracy/cost trade-off, developers typically have to tune a large number of model parameters (such as embedding widths, layer sizes, etc.). Even with a small number of options for each hyperparameter, the search space grows combinatorially and quickly becomes intractable, even for iterative architecture search methods [14]. To make the search cost-effective, we use TuNAS, a neural architecture search algorithm based on weight sharing [15]. This allows us to efficiently explore high-dimensional network configuration spaces.

To use TuNAS, model developers need to convert a model into a *search* model with candidate search space, e.g. a set of candidate widths for each hidden layer, and train the model to collect search result. After the search is done, developers need to bake in the search result, which contains the concrete width for each embedding or hidden layer, to the original model and retrain a *static* model.

D. Vizier

Vizier [8] is a state-of-the-art black-box optimization service used for hyperparameter tuning. It provides a uniform interface (RPC service) for users to conduct hyperparameter tuning studies. Although Vizier studies are more expensive than TuNAS, non-architectural hyperparameters (specifically, those globally affecting the learning process, such as learning rates, input data, batch size, etc.) can *only* be tuned with Vizier.

To use Vizier, developers need to build an application specific hyperparameter tuning system on top of the Vizier service. The tuning system creates a Vizier study with a search space, a search algorithm and a set of parameters. Then, Vizier provides *trials* setting concrete hyperparameter values, and the tuning system bakes in those hyperparameter values into the model, evaluates the model performance, and reports the result to the Vizier service. Based on these results, Vizier proposes new trials until the optimization is complete.

III. Implementation

The HEINZELMAENNCHEN system performs end-to-end training and evaluation for users. To begin such an *experiment*, a user specifies the *target model* to base the experiment on, the *protocol* to apply, and the modeling *intent*.

HEINZELMAENNCHEN maintains a database of available protocols. Each protocol encodes a specific modeling task, for instance, tuning the learning rate of a model. Protocols are discussed in detail in Sec. III-D.

The training service maintains a list of all available models, which are indexed simply by name. HEINZELMAENNCHEN

maintains a database of metadata associated with these models, containing information needed for optimization. This includes some high-level information about the model architecture that is useful for automation, but most importantly, it tells the system which metrics are the best predictors for launch readiness, and which thresholds to apply. For example, the HEINZELMAENNCHEN system knows, for each model, which is the primary metric to optimize for, and what amount gain in this metric is required to create a viable launch candidate (candidate models with marginal gain may not be launched, the return on the amount of work is too small). It also contains launch constraints regarding the model size (RAM requirements) or speed (indicative of training cost). These constraints are used to guide the TuNAS or Vizier optimizations, and to report whether a given experiment was successful, i.e., whether a viable launch candidate was found.

Several separate sets of constraints can be defined, each associated with a modeling *intent*. The two most common modeling intents are *improve quality* and *optimize resources*.

Gathering and encoding this information happens during *model onboarding*. We provide a Python library which allows model developers to include metadata directly in the model. Alternatively, annotations can be added to a database, either for a single model, or for a class of models.

To start an experiment, a user selects a target model, a protocol, and a modeling intent. We provide a CLI as well as a Web UI which guides users through the process. To support bulk processing, we offer mechanisms to apply a set of protocols to a set of target models. For details on the available API, see Sec. III-A. The system uses the user-provided protocol, target model, and intent to create an *experiment configuration*, which controls the experiment behavior. It is possible to edit this configuration to override defaults or details inferred from the stored metadata.

Upon confirmation, the service creates one or several experiments, which are added to a queue awaiting execution. Once resources are available, and constraints such as rate limiting allow an experiment to proceed, the experiment is scheduled, and its protocol starts to execute.

At the heart of each protocol is a *recipe* (or several), a program which modifies a copy of the target model to create a *proposed model*. See Sec. III-E for details on recipes. While some protocols can be rather complex, the simplest and most common protocols apply their recipe to create a proposed model, and then train and evaluate the proposed model against the constraints provided in model onboarding. Details on protocols are found in Sec. III-D.

When an experiment finishes, the system sends the owning user an email notification with the experiment result and a link to a status page. If the proposed model is promising, users can start live A/B testing, and eventually launch the modeling change if it meets the launch criteria.

A. API

The system provides two groups of RPC endpoints. For convenience, these RPCs are exposed via a CLI and a Web interface, but we also make these RPCs public to allow advanced users to control the system programmatically.

The first group of RPCs allows users to manage *registrations*, which cause the system to generate experiments. The second group deals directly with the generated experiments.

The main entry point for users is the `RegisterModels` RPC. It allows users to provide one or more models, and a list of protocols along with experiment configurations. Experiment configurations can be automatically generated from modeling intents. The system stores these lists and will schedule experiments for the cross product between all given protocols and models. It returns a registration ID which can be used to cancel the registration (`CancelRegistration`) or query its state (`GetRegistration`). Cancelling a registration also cancels all experiments that have been scheduled.

Users can also manage experiments directly, using `SearchExperiments` to find experiments using a search query, `GetExperiment` to view details on an individual experiment, and `CancelExperiment` to cancel a single experiment.

This small set of primitives is sufficient to provide the functionality needed. In particular, explicitly allowing for bulk processing is important to enable working efficiently with large numbers of models.

B. Model Onboarding

When working with a large population of models maintained by hundreds of ML engineers, any individual modeler will not know the details of every model. In order to allow for automated changes and evaluation of experimental results, we have to store metadata about models.

Users can add such metadata directly to the model code. This has the advantage that such annotations are carried over automatically when a model is branched and modified.

Model metadata added in this manner needs to be robust to typical modeling changes. This ensures that the model onboarding process is only a one-time cost to users. In cases where annotations have to be changed due to changes to the model itself, we found that embedding the annotations in the model code successfully nudges developers to update them along with their other changes.

Metadata that is typically added to model code includes high-level structural information, for example, the name of the first dense layer, the most important model tower, or the main loss objective. This type of annotation makes structure-modifying protocols easier to implement and more robust, as they otherwise have to infer this information by analysing the model's computation graph.

Another type of annotation allows for configuration of the evaluation data to use. For example, some models should only be evaluated on data from a specific date range. Without this information, the automatically produced evaluations might be unreliable and misleading.

Most importantly, models define evaluation metrics for debugging and performance analysis. In order to automatically determine whether a candidate model is promising, the system needs to know which metrics are important, and how much

they need to improve to make a change worthwhile. We therefore allow defining a set of *constraints*, each of which specifies a metric, an expectation indicating whether the metric should be smaller or larger, and an absolute or relative threshold.

Some protocols, especially TuNAS or Vizier-based protocols, can optimize a wide variety of metrics. In such cases, users have to communicate their modeling *intent*, or, which metric they want to consider the primary optimization objective. Such a modeling intent is expressed as a named set of constraints. We have found that these modeling intents are reusable across many models, and we therefore store them in a central database instead of embedding them with the models.

The most common intents are *improve quality*, which is encoded as a threshold requiring a certain improvement of a quality metric while keeping performance metrics neutral, and *optimize resources* which requires an improvement in one or more performance metrics (e.g., RAM utilization, training step time), while keeping quality metrics neutral. Users can pick one of the pre-defined intents by its name when they start an experiment. If an explicit intent is absent, the system infers the most likely one from the type of protocol used.

C. System Architecture

Our system is built as a collection of microservices (see Figure 1): Our system's user-facing components are a command line interface (CLI) and a Web UI, both of which communicate, via a public API, to an API server. The UI is served from a separate frontend server. A scheduling server schedules experiments. Once an experiment starts, it may create one or more workflows, whose execution is orchestrated by a dedicated orchestration server. The services are backed by a Spanner database [16], which persists all necessary system state to provide robustness against machine failures, and which provides queuing primitives with strong execution guarantees, significantly simplifying the service logic. Auxiliary services perform garbage collection (workflow cleaner), send emails and notifications (email service) and snapshot the experiment database for analysis (experiment exporter).

1) Web UI & CLI: Users interact with the service either via a CLI or a Web UI provided by the frontend server. The UI is designed to be experiment centric, because we find that it is more intuitive for users to think in terms of experiments. The UI provides a launch page, a search page, an experiment page and an analysis page. Figure 2a shows the launch page, which allows users to start an experiment given a target model, a protocol, and an intent. If the model is correctly onboarded and sufficient metadata is available to the system, the system can fully populate the experiment configuration. Users can override or if necessary, complete the configuration. Some protocols are parameterized, and users can edit such protocol parameters. Many protocols support parameters that can be chosen by automatic optimization (i.e., using Vizier), or which can be manually specified (e.g. controlling the downsampling rate for input data reduction).

The search page shows all experiments matching a query. Most common queries include searching experiments by experiment owner, target model sub-string or protocol name. Clicking any experiment on the search page navigates to the an experiment page, which shows the model registration and experiment details. It allows users to check experiment state, and for finished experiments, analyze the result. Figure 2b shows the analysis page, which allows comparing evaluation metrics of multiple experiments that share the same target model and protocol, but with different protocol parameters.

The Web UI enables a much smoother user journey and is important for broad adoption of the system. In total, 94% users are using the Web UI. They have created 76% of all experiments. The CLI is used for scripting and bulk experiment creation.

2) API Server: While most users use the CLI or Web UI to interact with the system, we make the API server publicly accessible, and it can be used directly by advanced users. It exposes the public API discussed above, allowing users to manage registrations and experiments. We use Spanner queues to perform work on demand, and to guarantee that execution happens exactly once. The listener API endpoints are triggered by such queue events. This design pattern allows us to offload consistency concerns to Spanner. Even in the presence of multiple API server instances, we do not need to implement custom logic to ensure events are processed exactly once.

For example, to monitor running experiments, the `LeaseExperimentListener` is periodically triggered by Spanner messages. The handler communicates with the orchestration server to check if the workflow executes normally. If so, another message is put into the queue which will trigger the same handler again in the future. If the workflow has failed, the `LeaseExperimentListener` marks the experiment as `FINISHED`, potentially triggering cleanup work. Other listener endpoints work similarly.

The `LeaseExperimentListener` also handles the majority of error classification and reporting. In case a workflow fails, we attempt to communicate a precise reason to the user. For bookkeeping, we distinguish among failures due to invalid user inputs, in which case the `LeaseExperimentListener` sets the experiment status to `USER_ERROR`. If the failure is due to a system error which requires our investigation, the status is set to `SYSTEM_ERROR`. Essentially, workflow execution failures that are not user errors are classified as system errors. To improve system reliability, we want to minimize the percentage of experiments that end with system errors. If the workflow is stuck, the `ExpireExperimentListener` sets the experiment status to `DEADLINE_EXCEEDED`. If users cancel an experiment, its status is set to `CANCELED`.

3) Scheduling Server: When an experiment is created in response to a registration, it starts in the `ENQUEUED` state. The scheduling server decides which enqueued experiments to launch (transitioning them to `IN_PROGRESS`). Once an experiment launches, a corresponding *workflow* is created, which is then managed by the orchestration server. Conceptually separating workflows and experiments allows us to create experiments quickly in the API server, while starting an experiment by creating its workflow could take minutes.

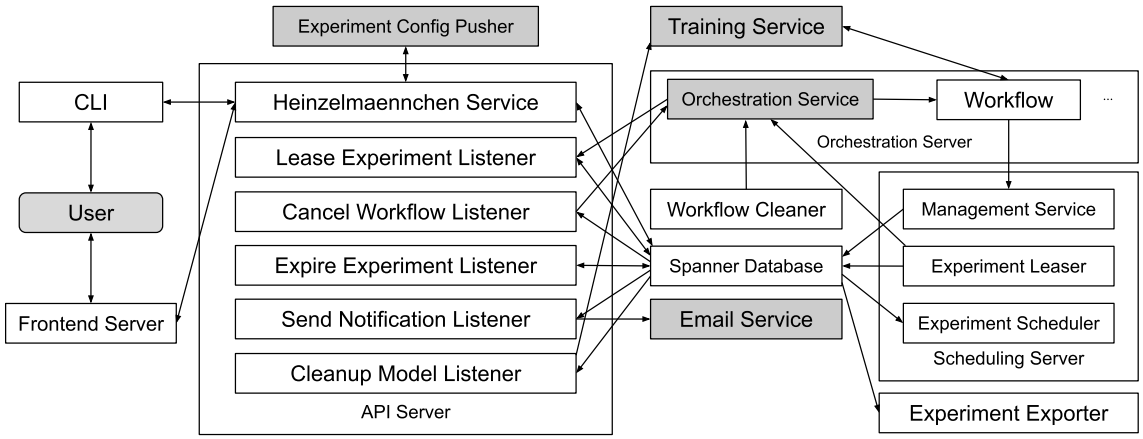
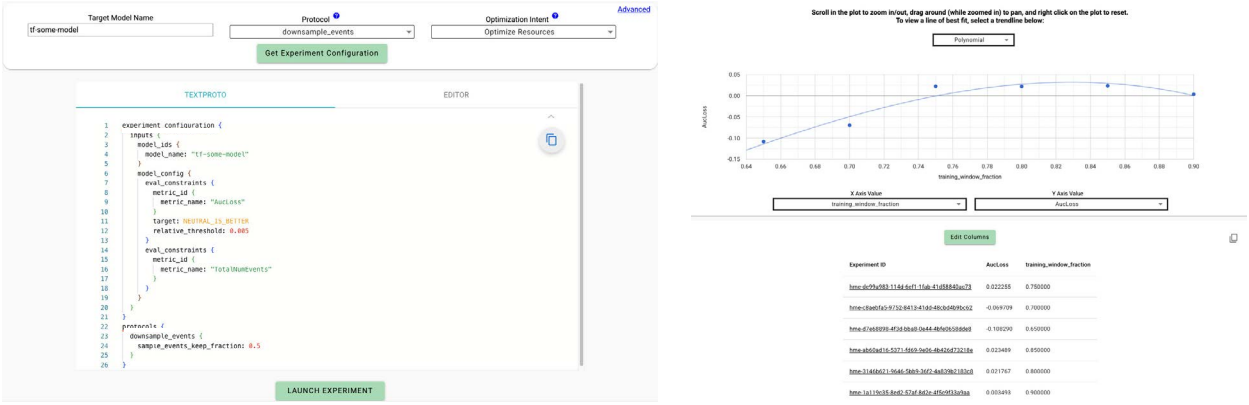


Fig. 1: Overview of the system architecture. Shaded boxes are external services. Arrows denotes data flows.



(a) Experiment launch page

(b) Experiment analysis page

Fig. 2: Example Web UI pages.

The scheduling server provides a quota mechanism for resource management: experiments are associated with a *quota group* based on the target model's owning team.

The `ExperimentScheduler` monitors the number of `IN_PROGRESS` experiments for each quota group. It randomly picks a quota group that is not overloaded with `IN_PROGRESS` experiments, finds `ENQUEUED` experiments in that quota group, and marks those experiments as `IN_PROGRESS`. This makes sure that the number of `IN_PROGRESS` experiments for each quota group is always below the allotted quota.

When an experiment is scheduled, the system asks Spanner to trigger `LeaseExperimentListener` after 5 minutes and `ExpireExperimentListener` after 30 days, respectively. Then, the `ExperimentLeaser` invokes the `CreateWorkflow` RPC to launch a workflow and persists the returned workflow ID to the database. To avoid `IN_PROGRESS` experiment hanging forever when its workflow creation is interrupted, e.g. when the server is shutdown for maintenance, we give `ExperimentLeaser` 5 minutes to store the workflow ID. If the first invocation of `LeaseExperimentListener` cannot find a workflow ID, it resets the experiment to `ENQUEUED` for rescheduling.

The scheduling server provides the experiment `ManagementService` exposing the internal `FinishExperiment` RPC endpoint. At the end of the workflow execution, it invokes the `FinishExperiment` RPC to store experiment results and sets the experiment state to `FINISHED` in the database. If the proposed model satisfies all evaluation constraints, the system sets the experiment status to `SATISFIED`. If any constraint is not satisfied, the status is set to `UNSATISFIED`.

4) Orchestration Server: The orchestration server is responsible for managing the execution of the protocol. It takes as input a workflow config, which contains a graph of steps. Each step is a binary that can take inputs and store outputs on a remote file system. A step output can be used as another step's input, which introduces a step dependency. All steps form an acyclic graph and the orchestration server remotely executes these steps in topological order, making sure that a step is only executed when all its inputs are ready. The orchestration maximizes step execution parallelism to minimize workflow end-to-end latency. The orchestration service provides an API allowing the API server's `LeaseExperimentListener` handler to find failed steps and report errors.

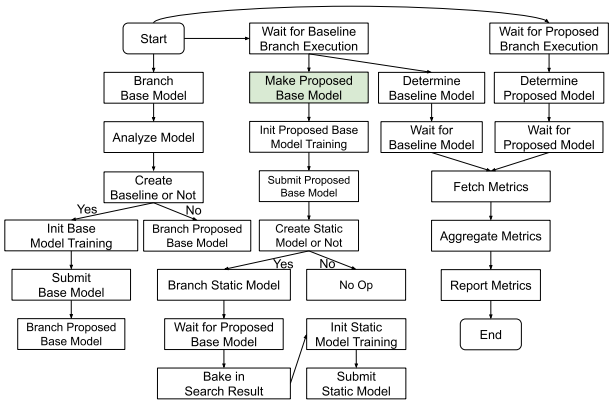
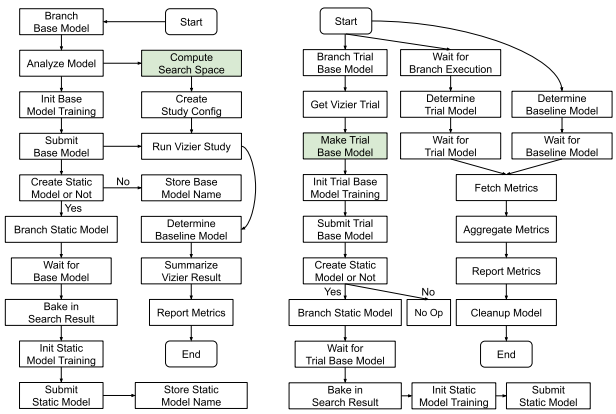


Fig. 3: Workflow of a basic protocol.



(a) Create Vizier study (b) Evaluate Vizier trial

Fig. 4: Workflow of a hyperparameter tuning protocol.

5) Workflow Cleaner: The workflow cleaner periodically cancels running workflows that are created sufficiently long ago, since they are likely stuck or were orphaned during experiment scheduling. For example, if it takes >5 minutes to create a workflow, the `LeaseExperimentListener` would incorrectly assume that the workflow failed to launch, then mark the experiment as `ENQUEUED` again to allow a scheduling server to reschedule the experiment. We garbage collect these workflows to avoid zombie experiments accumulating and wasting resources.

6) Experiment Exporter: The experiment exporter periodically reads all finished experiments in the past day from the database and exports these experiment records to a separate datastore for data analysis [17] and persistence. It is critical to help us understand the system usage and collect failed experiments for triage. To avoid database read contention, the job is scheduled daily early morning when the database has low read traffic.

D. Protocol

Protocols are implemented as orchestration workflows and they carry out the actual work of the system. To allow customizing modeling changes, protocols can have parameters and users can update these parameter values in the model registration via our CLI or Web UI. We observe that most protocols developers have created conform to one of two templates: a *basic* protocol, and a *hyperparameter tuning* protocol. We have standardized the implementation of protocols to inherit from these templates whenever possible, which significantly increases code reuse and reduces maintenance effort. Note that the system is not restricted to these two templates, developers are free to define any protocol.

1) Basic Protocols: Figure 3 shows a `Basic` protocol. The protocol branches a base model from the target model, compiles and analyzes the base model to extract information used in subsequent steps, then checks if it needs to retrain a baseline model. If so, it initializes and submits the base model for training, and this new model will be subsequently used as the baseline; otherwise the target model is used as the baseline. Retraining the target model may be needed when the target model is very old, which could make performance evaluation inaccurate after recent training service binaries releases and other changes in the software stack. In either case, the protocol branches from the baseline model and executes the `ApplyRecipe` step to apply a change/recipe to the new model, which creates the proposed model (highlighted in green in Fig. 3). The protocol then initializes and submits the proposed model for training. After both the baseline and proposed models are have been created, the protocol waits until both models complete training. It is possible for the `ApplyRecipe` step to create an TuNAS search model as its output. In case where TuNAS is involved, metrics measured on the model used for the hyperparameter search are unreliable [15]. We therefore create another model, bake in the best parameters identified in the search, and train this static model as the proposed model. This means that the cost of a TuNAS-based hyperparameter search is roughly twice the cost of a regular model training run.

Finally, after the baseline and proposed models are fully trained, the protocol fetches and aggregates the metric values, and reports the metrics back to the scheduling server.

The `ApplyRecipe` step is the main way users customize basic protocols: Most basic protocols reuse all steps but implement their own `ApplyRecipe` step. This step performs the actual edits to the model code. We discuss details on how we achieve robust modifications of the model in Sec. III-E.

2) Hyperparameter Tuning Protocols: Figure 4 shows a protocol for model hyperparameter tuning. It consists of two workflows: `CreateVizierStudy` and `EvaluateVizierTrial` shown in Figure 4a and Figure 4b, respectively.

The `CreateVizierStudy` workflow branches a base model from the target model, then initializes and submits the base model. The `ComputeSearchSpace` step (highlighted in green in Figure 4a) computes the hyperparameter search space based on the target model and protocol inputs. This is the main way the `CreateVizierStudy` workflow differs for various use cases.

Next, the template creates a Vizier study given the computed search space. The `RunVizierStudy` step maintains multiple

Vizier clients to orchestrate trials in parallel. Each Vizier client initiates an `EvaluateVizierTrial` workflow to evaluate the trial, using the base model as the baseline to compute the performance delta in percentage of the trial model. Once the Vizier study is complete, the `CreateVizierStudy` workflow retrieves the best trials and their metric values from the Vizier server. Finally, the protocol reports the result back to the scheduling server.

The `EvaluateVizierTrial` workflow functions similarly to a basic protocol described above, with a specific `ApplyRecipe` step (highlighted in green) that applies the concrete hyperparameter values suggested by Vizier for this particular trial to the model. It differs from the basic protocol in the way that it handles the trial model metrics, since it has to fetch aggregated metric values and report the results to the Vizier server. This allows Vizier to decide the next trials based on previous trial results. At the end of each trial, the workflow cleans up the trial model and frees up training resources, which is essential as hyperparameter tuning protocols can create many models, and have a tendency to quickly occupy huge amounts of resources.

The `ComputeSearchSpace` and `ApplyRecipe` steps are the main customization points for hyperparameter tuning protocols. Different hyperparameter tuning protocols share all other steps. The `ComputeSearchSpace` step extracts the hyperparameter search space from user input. For example, some protocols allow users to set the search space explicitly in their parameters while other protocols allow users to declare the search space in the model construction plan. The `ApplyRecipe` step takes as input a trial with concrete hyperparameter values, and updates the corresponding hyperparameters in the model construction plan or training job configuration.

The hyperparameter tuning protocols also provide for a hierarchical search strategy, where the protocol first searches some hyperparameters using Vizier, and each trial model is itself a tunable model using TuNAS, as described in [18]. Both `CreateVizierStudy` and `EvaluateVizierTrial` templates are able to handle baseline/trial models which are TuNAS search models in similar to the basic protocol: when an TuNAS search model is detected, when training finishes, instead of using the metrics generated by the search model, the search results are extracted and baked into a static model which is trained to obtain the final metrics for the baseline/trial models. The hierarchical search strategy is a trade-off between the search cost and flexibility. It is more resource efficient compared to using Vizier alone while solving problems for which TuNAS is not applicable. A simple example of a hybrid protocol is tuning the network depth (number of layers) with Vizier while tuning the network size (layer widths) with TuNAS.

E. Recipe

It is difficult to build a general purpose code editing tools to modify the model construction plan. Like most programs in the machine learning space these days, model construction plans are written in Python. As a dynamically typed language, modifying Python code safely is particularly challenging. The problem is compounded by variability in coding style between different teams, making it dangerous to assume common patterns for code manipulation.

(A) High level model rewrite.

```
1.+ def UpdateLearningRate(model, lr):
2.+   model.optimizer.learning_rate = lr
3.  inputs = keras.Input(...)
4.  outputs = keras.layers.Dense(...)(inputs)
5.  m = keras.Model(inputs=inputs, outputs=outputs)
6.  m.compile(loss="mse", optimizer="adam")
5.+ UpdateLearningRate(m, lr=0.1)
```

(B) Function call interception.

```
1.+ def DoubleEmbeddingDim(in_dim, out_dim, **kwargs):
2.+   return keras.layers.Embedding(
3.+     in_dim, out_dim * 2, **kwargs)
4.  inputs = keras.Input(...)
5.- embedding = keras.layers.Embedding(
6.+ embedding = DoubleEmbeddingDim(
7.    vocab_size, output_dim)(inputs)
8.  ... // Rest of the model.
```

(C) Tunable API.

```
1.+ tunable.Set({
2.+   "width": 64,
3.+ })
4.  ... // Rest of the model.
5.  width = tunable.Int(
6.    name="width", default=32, candidates=[32, 64])
7.  outputs = keras.layers.Dense(width)(...)
8.  ... // Rest of the model.
```

Fig. 5: Example recipes.

We implemented several approaches to make systematic code edits for applying modeling changes. The design of our system allows us to pick the most appropriate `ApplyRecipe` step implementation, no single implementation has to be fully general. Figure 5 shows examples of the types of code modifications `ApplyRecipe` implementations perform. We use the Keras model-building framework for illustration purposes.

The most direct way to implement a recipe is to define a rewrite function that makes modifications to the high level model. The `ApplyRecipe` step then simply inserts the rewrite function definition, and calls it the end of the model construction plan, possibly using protocol-provided parameters. The function then modifies the Keras `Model` object in place. Because we make modifications at the end of the file (and therefore after the `Model` is fully defined), we can be sure that our modifications is final and will not be clobbered by other parts of the code.

Figure 5 (A) shows an example that modifies model code to update the optimizer learning rate using the `UpdateLearningRate` function. Such a rewrite function typically takes as input a `Model` and a list of parameters, and modifies the `Model`. In practice, we do not add the full definition of the rewrite function to the model definition, but simply import it from a public library.

Using rewrite functions has some limitations: While the rewrite is guaranteed to produce a working model, it is not considered particularly readable, and cannot be launched as is. If an experiment identifies a promising model, a human has to inline the rewrite into the model definition, e.g., changing the learning rate in the `compile` call where it was originally defined. Another limitation lies in the complexity of modifying `Model`s: if parts of the model we want to update are used to build other parts of the model, it can be difficult to implement

the recipe in a rewrite function. For example, scaling the embedding layer dimensions could change the input shape of subsequent layers, which requires the rewrite function to properly update all transitively dependent layer properties.

Another method for modification is function call interception. Recipe developers define a function meant to replace a common library function used in the model definition. The `ApplyRecipe` step replaces all library function calls with calls to the new function. For simplicity, the replacement function has the same signature as that of the common library function to replace with. We use an code AST representation [19] to robustly identify and edit call sites. Figure 5 (B) shows an example that doubles the embedding dimension of all features using the `DoubleEmbeddingDim` function, which intercepts the `Embedding` initializer call. Since the embedding dimension is updated in-place, all subsequent layers are built using the updated input shape automatically.

Not all model changes can be implemented using function call interception, for instance, the learning rate in Figure 5 (A) is implicitly set so we cannot overwrite it via function call interception.

If users are willing to annotate their model, we can automate richer modeling changes. For example, our system provides a `tunable` API that allows users to define hyperparameters with search space anywhere in a model, including inside a libraries transitively invoked in the model. The hyperparameter and its search space is then stored in the high level model. In the `CreateVizierStudy` template, the `ComputeSearchSpace` step can extract the search space from the high level model. In the `EvaluateVizierTrial` template, the `ApplyRecipe` step can set hyperparameter values in the model. Figure 5 (C) shows an example. Lines 4-8 is the original model in which a user defines an integer hyperparameter to tune the dense layer width. The hyperparameter has a name called `width`, a default value 32 and a list of possible values [32, 64] as the search space. The `ApplyRecipe` step inserts a `tunable.Set` function call at the beginning of the trial model (lines 1-3) to set the `width` hyperparameter to 64 in a global registry. `tunable.Int` checks this global registry using the hyperparameter name, and returns 64. We have found that because `tunable.Int` can return a default value without changing the model semantics, users are willing to annotate their models with hyperparameters proactively and use HEINZELMAENNCHEN to tune hyperparameters when needed.

IV. RESULTS

A. Production Setup

HEINZELMAENNCHEN is deployed geographically at 3 locations in the US and each location has 3 API servers, 3 scheduling servers and 3 frontend servers that serve traffic all over the world. This distributed setting avoids a single point of failure. The workflow cleaner and experiment exporter jobs are executed on a daily basis. The paper reports the performance of the system using production data; statistics have been collected from 04/01/2023 to 10/01/2024 (1.5 year).

TABLE I: API request count and latency.

API	Count	Latency (ms)				
		Avg	p50	p90	p99	Max
RegisterModels	6562	848	742	1311	3127	39688
GetRegistration	30206	45	17	148	305	2689
CancelRegistration	233	695	600	1360	1360	1372
SearchExperiments	64022	111	76	1254	787	14464
GetExperiment	20443	89	53	145	671	5271
CancelExperiment	540	1115	1017	1805	4789	7379
FinishExperiment	4348	872	777	1372	2822	10331

TABLE II: User, experiment and model count by status.

API	#User(%)	#Exp(%)	#Model(%)
SATISFIED	104 (50.7)	1461 (15.1)	8382 (9.2)
UNSATISFIED	117 (57.1)	1402 (14.5)	8358 (9.2)
USER_ERROR	181 (88.3)	3689 (38.2)	4281 (4.7)
SYSTEM_ERROR	87 (42.4)	566 (5.9)	15142 (16.6)
DEADLINE_EXCEEDED	58 (28.3)	373 (3.9)	6434 (7.1)
CANCELED	130 (63.4)	2157 (22.4)	48664 (53.3)
Total	205 (100)	9648 (100)	91261 (100)

B. System Performance

Table I shows the number of requests and latency for each RPC. The `RegisterModels` RPC had 6562 requests and created 9648 experiments, with an average latency of 848ms. Typically requests that create multiple experiments require more time to process compared to requests that create a single experiment. The `RegisterModels` RPC is also more expensive compared to other RPCs because it writes more data to the database. The `GetRegistration`, `SearchExperiments` and `GetExperiment` RPCs are the main user-facing endpoints and therefore commonly used. The `SearchExperiments` RPC backs the experiment search page, which is the most visited UI page. All three RPCs are relatively cheap because they only read the database. The database-modifying `CancelRegistration`, `CancelExperiment`, and the internal `FinishExperiment` RPCs are used much less, and they are slightly more expensive. The average latency for all RPCs is less than 1.2 seconds.

Table II shows the number of users, experiments and models by experiment status, respectively. In total, the system was used by 205 users (46.6 monthly), and it created 9648 experiments (26.4 daily) and 91261 models (250.0 daily). The system found promising candidate models from `SATISFIED` experiments for 104 users. 117 users encountered `UNSATISFIED` experiments. Note that `UNSATISFIED` experiments are typically still useful, often because they disprove a hypothesis.

15.1% of experiments resulted in a `SATISFIED` status, meaning they created viable launch candidates. In our experience, this compared very favorably to the success rate of manual experimentation. We credit the fact that we encode only approaches into protocols that are likely to work broadly. This leads to some selection bias, but it also legitimately improves the success rate of experimentation, as developers create fewer futile experiments.

38.2% experiments ended with `USER_ERROR`, as it takes time for users to learn how to use different protocols. Most `USER_ERROR` experiments are caused by user requesting metrics that do not exist, or incorrectly annotated model components.

TABLE III: Median experiment time breakdown by status.

Status	Queuing	Scheduling	Execution
SATISFIED	1.1s	102s	46h
UNSATISFIED	1.1s	131s	73h
USER_ERROR	1.1s	141s	8h
SYSTEM_ERROR	1.1s	125s	5h
DEADLINE_EXCEEDED	1.1s	118s	719h
CANCELED	1.2s	118s	18h
Median	1.1s	125s	26h

Only 5.9% experiments ended with SYSTEM_ERROR, demonstrating the reliability of the system. Common SYSTEM_ERROR experiments are caused by newly implemented protocols with bugs, or system reliability issues like exhausting orchestration resources. 3.9% experiments ended with DEADLINE_EXCEEDED, which is most often caused by experiments never being allocated enough resources to train models.

Table III shows the median experiment time overhead in queuing, scheduling and execution phases, respectively, by experiment status. Almost all ENQUEUED experiments were scheduled immediately (median at 1.1s) as scheduling these experiments rarely cause the number of IN_PROGRESS experiments to exceed the pre-defined quota limit. The median time to schedule an experiment and create a workflow is 125s. The scheduling latency is dominated by the CreateWorkflow RPC latency, which is large as the orchestration server takes time to parse and validate the workflow config. The execution time varies significantly for experiments ending with different statuses. The median execution time for SATISFIED and UNSATISFIED experiments is roughly similar, the only difference being the result of constraint evaluation. DEADLINE_EXCEEDED experiments take a long time to complete by definition, they are expired after 30 days by default. This explains why the median execution time 719h is close to 720h (30 days). Experiments ending with errors (USER_ERROR or SYSTEM_ERROR) often finish quickly, as the system attempts to detects and report errors as soon as possible.

C. Protocol Performance

Table IV shows the experiment stats of the top 10 widely used protocols. $\#\text{Exp}_t$ denotes the total number of experiments. $\#\text{Exp}_r$ denotes the number of experiments with result (SATISFIED or UNSATISFIED), and % denotes the percentage of experiments with result compared to all experiments. $\#\text{Exp}_s$ denotes the number of SATISFIED experiments, and % denotes the percentage of SATISFIED experiments compared to all experiments. #Model denotes the total number of generated models. #M/#Exp denotes the number of generated models per experiment on average. Exec_r denotes the average execution time of experiments that are finished with a result.

The set of protocols supported by the system is vast and still growing. The ten most widely used protocols cover feature engineering, architecture search, hyperparameter tuning, and data engineering.

ReduceTrainingData is a simple protocol that reduces the amount of training data used. It is one of the simplest yet most successful techniques to save training resources.

TrainModel is a simple protocol that retrains a model, which is common when users want to check model reproducibility or debug a model.

TuneHyperparameters is a Vizier protocol that tunes arbitrary hyperparameters in the model. It is the most widely used protocol in the system due to its generalizability. SetHyperparameters allows users to manually set arbitrary hyperparameters in the model, which works well when the search space is small.

TuneShampooOptimizer replaces the existing optimizer with Shampoo [20], and then tunes hyperparameters. ApplyMoE applies a mixture of experts architecture [21] to a fully connected network, and then tunes the hidden layer widths.

In terms of feature engineering, TuneEmbeddingWidths is a TuNAS protocol that tunes the embedding dimensions of the model. It is used for embedding dimension optimization or feature selection (it can, and will, reduce embedding dimensions for some features to 0, effectively removing the feature from the model). UpdateFeatures can more explicitly add, replace or remove features from a model, while AddDCN adds a deep cross network [22] to learn feature crosses automatically.

TuNAS and hyperparameter tuning protocols have more internal steps and require complex coordination between different services. They are therefore less reliable: a smaller percentage of experiments ends with SATISFIED or UNSATISFIED statuses compared to other protocols. There are some exceptions, e.g. TuneShampooOptimizer is used by few expert users, and 49.8% experiments finish without errors, mostly because this protocol's main users appear to make fewer user errors.

Each Vizier protocol execution often create multiple trial models, hence there are many more models associated with these protocols. TuneHyperparameters alone created 82012 models (70.88 per experiment), and TuneShampooOptimizer created 1434 (6.49 per experiment). TuNAS protocols on the other hand create only one search model and one static model, leading to much fewer models compared to Vizier protocols. For example, TuneEmbeddingWidths created 1187 models (0.82 per experiment), and ApplyMoE created 534 models (0.79 per experiment). Regular protocols typically only create a single model, leading to the least number of models, e.g. ReduceTrainingData created 0.67 models per experiment, and TrainModel created 0.46 models per experiment.

These numbers are a stark reminder how computationally expensive black-box optimization techniques such as Vizier are. Vizier experiments also take a much longer time to complete due to the number of models they create (a limit on the maximum number of concurrently training trial models is enforced to avoid resource contention), e.g. TuneHyperparameter and TuneShampooOptimizer respectively take 180h and 183h to run successfully on average. TuNAS experiments take less time to complete but they still need to wait for the training completion of search models before they train static models, e.g. TuneEmbeddingWidths and ApplyMoE respectively take 105h and 154h to run successfully on average. Regular experiments often take the least amount of time to finish as they train 1-2 models with maximal parallelism, e.g. AddDCN

TABLE IV: User, experiment, model count and execution time for top 10 widely used protocols.

Protocol	Type	#User(%)	#Exp$_t$(%)	#Exp$_r$(%)	#Exp$_s$(%)	#Model(%)	#M/#Exp	Exec$_r$
TuneHyperparameters	Vizier	71 (34.6)	1157 (12.0)	370 (32.0)	174 (15.0)	82012 (89.9)	70.88	180h
TuneEmbeddingWidths	TuNAS	63 (30.7)	1449 (15.0)	373 (25.7)	114 (7.9)	1187 (1.3)	0.82	105h
ApplyMoE	TuNAS	40 (19.5)	673 (7.0)	196 (29.1)	51 (7.6)	534 (0.6)	0.79	154h
TrainModel	Regular	34 (16.6)	2324 (24.1)	829 (35.7)	430 (18.5)	1059 (1.2)	0.46	78h
ReduceTrainingData	Regular	30 (14.6)	280 (2.9)	132 (47.1)	48 (17.1)	187 (0.2)	0.67	107h
TuneShampooOptimizer	Vizier	24 (11.7)	221 (2.3)	110 (49.8)	41 (18.6)	1434 (1.6)	6.49	183h
UpdateFeatures	Regular	23 (11.2)	391 (4.1)	162 (41.4)	61 (15.6)	252 (0.3)	0.64	118h
SetHyperparameters	Regular	17 (8.3)	567 (5.9)	208 (36.7)	66 (11.6)	375 (0.4)	0.66	81h
AddDCN	Regular	17 (8.3)	39 (0.4)	17 (43.6)	6 (15.4)	28 (0.03)	0.72	50h
TuneTpuEfficiency	Vizier	12 (5.9)	222 (2.3)	75 (33.8)	40 (18.0)	2281 (2.5)	10.27	121h

and `TrainModel` take 50h and 78h, respectively, to run successfully on average. In practice, other factors can also affect the successful experiment execution time. For example, some models are faster to train, and some models can be queued for lacking training resources.

D. System Impact

Over the evaluation period, the system was used actively by more than 50 teams, contributing to more than 50 model launches. These launches included quality improvements, resource savings, and infrastructure migrations.

The developer velocity improvement is hard to measure, so we interviewed tech leads whose teams used our system and collected their feedback. Our system is particularly effective for teams responsible for more than a handful of models. Manually evaluate the a technique on all models is tedious and time-consuming, and automation dramatically increases productivity. For example, Team A, which is responsible for maintaining more than 70 production models, saw 5x the number of feature engineering and network architecture launches after adopting the HEINZELMAENNCHEN. Another team, maintaining more than 40 production models, emphasized that the system accelerates their model launches and frees up developer time.

Some teams own few but very important models. These models are particularly hard to improve as many researchers/developers have been optimizing these models for years. Those teams are surprised to see that our system can still improve their model quality and training efficiency. For example, hyperparameter tuning protocols with hierarchical search found network architecture changes in the most important Google Search Ads click-through-rate prediction model, which significantly reduces the training cost while keeping the quality neutral.

On the infrastructure side, the system contributes directly to developer productivity: The training service infrastructure team uses our system to canary their training job binaries by ensuring model re-trains are neutral when executed on new release candidates. The site reliability engineering team uses our system to perform training configuration flag tuning aiming to improve compute efficiency.

Small teams which lack the bandwidth or expertise to keep up with recent research results benefit especially from the knowledge that is encoded in the protocols and recipes. These teams can apply advanced techniques without having to adapt complex implementations to their models.

The ability to automatically find promising candidate models without writing model code saves developer time directly, e.g., by performing edits automatically, as well as indirectly, e.g., by avoiding errors and therefore reducing the number of edit/test cycles. Even `UNSATISFIED` experiments save model developer time. These are typically experiments that needed to be run to validate or falsify a hypothesis, which allows users to focus on other model optimization directions instead of spending much effort on non-promising directions.

V. Related Work

Researchers have developed a set of techniques to synthesize ML pipelines [10], [11], [23], [24], [25] from a pool of primitives, which includes different classifiers, data/feature preprocessing methods, and hyperparameters of these model techniques, etc. These techniques often assume that the problems are unsolved and the goal is to find the ML pipeline with the best accuracy. Their dataset is relatively small and the generated models are relatively simple. In comparison, our problems are challenging and many developers spend years to solve these problems. Our models are large and complex, trained on very large datasets. Thus, prior work cannot tackle problems at our scale. Another difference is that our system automates model development steps other than hyperparamter search, and these steps are common in industry, e.g. model initialization, submission, training and result analysis.

Google's Vertex AI [4], Amazon's SageMaker [3] and Microsoft's Azure ML [26] provide cloud ML solutions for individual customers. These works are analogous to the training service discussed in Section II-A. In comparison, HEINZELMAENNCHEN is built on top of the training service, which automates much richer ML modeling tasks at scale.

VI. Conclusion

In this paper, we propose HEINZELMAENNCHEN, the first ML modeling task automation system for production models at scale. The system is widely used by hundreds of ML engineers and creates many revenue critical launches. We also interviewed multiple teams and show that the system improves developer velocity. Although we build the system for Ads at Google, we believe that the system could provide useful insights in general for building similar automation systems.

Acknowledgments

We thank Ke Zhong, Andy Carlson, and the anonymous reviewers for their comments and feedback.

REFERENCES

[1] Iqbal H Sarker. Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, 2(6):420, 2021.

[2] Konstantinos Katsiapis and Kevin Haas. Towards ml engineering with tensorflow extended (tfx). In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3182–3182, 2019.

[3] Amazon sagemaker. https://aws.amazon.com/sagemaker/.

[4] Google vertex ai. https://cloud.google.com/vertex-ai/.

[5] Fblearner flow. https://engineering.fb.com/2016/05/09/core-infra/introducing-fblearner-flow-facebook-s-ai-backbone/.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[8] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[9] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[10] Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, Raoni de Paula Lourenco, Jorge Piazentin Ono, Kyunghyun Cho, Claudio Silva, and Juliana Freire. Alphad3m: Machine learning pipeline synthesis. *arXiv preprint arXiv:2111.02508*, 2021.

[11] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28, 2015.

[12] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[13] Apache airflow. https://airflow.apache.org/.

[14] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[15] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V Le. Can weight sharing outperform random architecture search? an investigation with tunas. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14323–14332, 2020.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[17] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[18] Sheng Li, Garrett Andersen, Tao Chen, Liqun Cheng, Julian Grady, Da Huang, Quoc V Le, Andrew Li, Xin Li, Yang Li, et al. Hyperscale hardware optimized neural architecture search. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 343–358, 2023.

[19] Pasta. https://github.com/google/pasta/.

[20] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, 2018.

[21] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 1991.

[22] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 2017.

[23] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[24] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 international conference on management of data*, pages 1171–1188, 2019.

[25] Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-Infante, Arun Ross, and Kalyan Veeramachaneni. Atm: A distributed, collaborative, scalable system for automated machine learning. In *2017 IEEE international conference on big data (big data)*, pages 151–162. IEEE, 2017.

[26] Azure ml. https://azure.microsoft.com/en-us/products/machine-learning/.