SMARTCHOICES: AUGMENTING SOFTWARE WITH LEARNED IMPLEMENTATIONS

Daniel Golovin [1] **Gabor Bartok** [1] **Eric Chen** [1] **Emily Donahue** [1] **Tzu-Kuo Huang** [1] **Efi Kokiopoulou** [1]
Ruoyan Qin [1] **Nikhil Sarda** [1] **Justin Sybrandt** [1] **Vincent Tjeng** [1]

ABSTRACT

We are living in a golden age of machine learning. Powerful models are being trained to perform many tasks far better than is possible using traditional software engineering approaches alone. However, developing and deploying those models in existing software systems remains difficult. In this paper we present SmartChoices, a novel approach to incorporating machine learning into mature software stacks easily, safely, and effectively. We explain the overall design philosophy and present case studies using SmartChoices within large scale industrial systems.

1 INTRODUCTION

Modern deep learning models power an increasing range of products and services, such as search, recommendation and discovery systems, and online advertising (among many others). However, training and deploying these models in production systems is fraught with new failure modes and opportunities to accrue distinct forms of technical debt (Sculley et al., 2015). Two major issues identified by those authors are that we cannot crisply define desired program behavior in cases where machine learning (ML) is necessary (which erodes abstraction boundaries) and most practitioners lack sophisticated tooling to track data provenance and data dependencies the way we do with source and object code.

In this paper, we re-envision the workflows to deploy machine learning in large scale systems, with an eye towards significantly reducing engineering effort and scope for errors. The result, *SmartChoices*, treats machine learning models as *learned implementations* within an application, which benefit from existing tooling for managing software. Ultimately, we aspire to make improving systems with ML nearly as easy as using a typical software library, and treat trained decision policies as code. Hence SmartChoices' design represents a fertile middle ground between grand ambitions to pervasively replace traditional software with deep learning, and the hard–won lessons of veteran engineers on how to build and run reliable production systems – and particularly systems that are critically dependent on machine learning models. It also means we diverge considerably from the design philosophy of ML platforms such as TFX (Baylor et al., 2017), Kubeflow[2], and others, which provide facilities to setup arbitrary ML pipelines that are not inherently tied to application behavior, are not tightly integrated into the client software, and require separate development workflows for pipeline management above and beyond standard software engineering workflows.

[1]Google Research. Correspondence to: SMARTCHOICES authors <smartchoices-2023-authors@google.com>.

2 SCOPE AND CAPABILITIES

We designed SmartChoices to address the following class of problems: A system is faced with a sequence of decisions, such that at time t it is provided a *context* $x_t \in \mathcal{X}$ as input, and a set of permissible outputs A_t (known as *arms* in the bandit literature) which is a subset of the universe of arms $\mathcal{A}$. It then must choose an arm $a_t \in A_t$ and receives feedback $y_t \in \mathbb{R}^k$ indicating the quality of the arm with respect to $k \geq 1$ metrics. The goal is to provide an implementation $\pi : \mathcal{X} \times 2^{\mathcal{A}} \to \mathcal{A}$ to optimize the metrics, which we call a *policy*. Throughout, we will refer to metrics we wish to maximize as *rewards* and those we wish to minimize as *costs*. In most cases, this implementation will be a parameterized function trained on available data $\{(x_t, a_t, y_t) : t \geq 1\}$. If $k = 1$, optimization consists simply of maximizing a reward or minimizing a cost. For $k > 2$, we consider two types of optimization tasks: metric-constrained optimization (e.g., maximize reward without increasing cost), and efficiently discovering the Pareto frontier of tradeoffs and then targeting a point along it selected by the system owner.

This problem class is broad and slightly underspecified. As such, we will illustrate the details in various special cases below and with case studies of real deployments in §6.

[2]https://github.com/kubeflow/kubeflow

2.1 Contextual Bandits

The most basic formulation for contextual bandits involves a fixed small universe of arms $\mathcal{A}$ (e.g., categories encoded as enum values) and a context domain $\mathcal{X} = \mathbb{R}^d$ for some d. SmartChoices supports several important modeling features beyond this basic formulation.

Mixed-type inputs SmartChoices supports mixed-type inputs (including numeric, enumeration, and string inputs) via automatically-generated embedding layers in our critic-model based implementation.

Case Studies: (i) deciding how much compute capacity (specifically, an integral thread pool size) to devote per task for a service with diverse task sizes (§6.3) and *(ii)* deciding whether to partition tasks for scheduling on multiple machines (§6.4).

Time-varying arm sets When $A_t \subset \mathcal{A}$, we enforce that $a_t \in A_t$ via selection masks in the policy.

Case Study: Selecting from a curated list of experiences (§6.6)

Arm-Features In many applications, the range of arms A_t is complex and may completely change over time (e.g., recommending today's top news stories). In such cases, it is critical to be able to generalize across arms. SmartChoices supports associating each arm with features (henceforth *arm-features*) to allow for generalization to unseen arms. For example, suppose there is an available embedding Φ of objects (e.g., sentences or images) into $\mathbb{R}^d$ for some d, and a retrieval process for selecting a reasonably sized set of candidate objects $C(x)$ for a context x. Then SmartChoices may select arms using the resulting embedding as features, by effectively choosing among $A_t = \{\Phi(c) : c \in C(x_t)\}$ and then returning the object associated with the selected embedding vector.

Case Studies: (i) improving the efficiency of a large content delivery network via a self-adapting cache (§6.1), *(ii)* accelerating ML workloads via smarter compiler optimization (§6.2).

2.2 Ranking

Ranking involves ordering some provided items and returning a list of the best ℓ of them. In this regard it involves a combinatorially large arm space (with $n!/(n-\ell)!$ possible arms given n items). It also involves a different feedback setting than contextual bandits, with feedback for particular positions (and hence items) in the list.

SmartChoices supports ranking via the use of a critic model $m_\theta : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ predicting the reward for each item. Users provide feedback via a scalar score for each list entry (henceforth *score vector feedback*); this enables us to support the popular *cascade click feedback* model, where we assume the user sequentially observes the items and interacts with the first one they find relevant.

In addition to a simple greedy approach that ranks items in descending order of predicted reward, SmartChoices allows for smart exploration via weighted sampling of items. Specifically, SmartChoices picks a list of ℓ items by sampling ℓ times from the Plackett–Luce distribution (Grover et al., 2019) seeded by the predicted reward. SmartChoices also supports diversity by penalizing items similar to the ones that have been already selected.

Case Studies: Ranking is used pervasively for recommendations. We discuss recommending actions for business owners to take to optimize their profiles (§6.7).

2.3 Multimetric optimization

In most applications, there are several metrics of interest, with inevitable tradeoffs.

2.3.1 Metric constrained optimization

A common response is to optimize with respect to one metric while constraining our decisions with respect to another. Examples include minimizing latency while maintaining a specified throughput target (latency vs. throughput), or maximizing the quality of results while keeping the average cost below a target (quality vs cost). SmartChoices enables metric constrained optimization to be done at the policy level. More specifically, we presume a distribution of decision problem instances (X, A) and an associated (possibly stochastic) scalar reward and cost(s) – all of which may be initially unknown – and a known vector of budgets C. SmartChoices can then search for implementations

$$\pi^* := \arg\max_\pi \mathbb{E}\left[\text{reward}(\pi(X, A))\right] \quad \text{s.t.} \\ \mathbb{E}\left[\text{cost}(\pi(X, A))\right] \leq C$$

Without contexts (i.e., $\mathcal{X} = \emptyset$), this is known as Bayesian optimization with unknown constraints (Gelbart et al., 2014). With contexts, this problem is very closely related to contextual bandits with knapsack constraints, for which there are known results for the stochastic (Agrawal et al., 2016) and adversarial settings (Sun et al., 2017). In contrast to the prior work, we are interested in per–instance average budgets (e.g., serving an infinite stream of online queries at bounded average latency) rather than cumulative budgets that eventually run out (e.g., dynamically pricing a limited supply of goods for maximum revenue).

An important consideration is that these constraints (specifically, the costs) must be learned. As such, the algorithm must be allowed to violate the constraints while learning. While there are ways to mitigate this (e.g., see §4.2),

SmartChoices is not designed or intended for circumstances where individual decisions are high-stakes (e.g., selecting medical treatments).

Case Study: We apply metric constrained optimization to choose when to update an ads cache, trading off engagement metrics with compute cost (§6.5).

2.3.2 *Pareto Frontier Search*

For applications with soft constraints, engineers may prefer a more exploratory approach than that of §2.3.1. In particular, we offer the ability to identify the set of possible tradeoffs via identifying the *Pareto frontier*, defined as follows. For a vector $y \in \mathbb{R}^k$, call it *achievable* if there is an implementation π with expected metrics y under the distribution of inputs and rewards. That is, $\exists \pi \,.\, \mathbb{E}\left[Y|\pi(X,A)\right] = y$. For maximization metrics (where higher values are desirable), the Pareto frontier is the set of achievable metric vectors $y \in \mathbb{R}^k$ such that no other achievable metric vector y' exceeds it in all metrics, i.e., satisfies $y'_i > y_i$ for all i.

To address this, SmartChoices supports *scalarizing* predicted metrics, i.e., combining them in a single scalar reward via a known *scalarization* function. Parameterized scalarization functions are supported via inference-time parameters, allowing the scalarization used to vary for each choice made, which enables efficient exploration of the Pareto frontier. Linear scalarizations (i.e., linear combinations of metrics) are simple and enable the discovery of Pareto frontiers when the achievable metrics form a convex set. For generally shaped Pareto frontiers, we use the hypervolume scalarizations (Zhang & Golovin, 2020), which can discover arbitrarily shaped frontiers.

Using scalarizations decoupled from the metric predictions has several advantages. It allows us to largely reduce the multiobjective optimization to the single objective case. This in turn enables us to reason about multiobjective optimization using theorems and algorithmic ideas developed for the single objective case. It also allows us to simplify our infrastructure. Finally, engineers can easily and rapidly focus on particular parts of the Pareto frontier, e.g., by adaptively selecting sets of scalarizations to experiment with in production and progressively narrowing in on regions of interest.

After investigating the Pareto frontier, engineers may elect to fix a tradeoff (i.e., scalarization parameters) corresponding to their preferred Pareto–optimal point and use it in all future policies. Alternatively, they may find it more natural to express desired system behavior in terms of metric constraints (§2.3.1). These options may appear equivalent at first glance, but they respond differently to changing input and reward distributions in important ways.

```
auto smartchoice = CreateSmartChoice<
    ExampleContext, ExampleArm, ExampleFeedback>();

ExampleContext context = ...;
ExampleArm chosen_arm;
auto feedback_handle = smartchoice.Choose(
    context, default_arm, candidate_arms, &chosen_arm);

ExampleFeedback feedback = ...;
feedback_handle.GiveFeedback(feedback);
```

Figure 1. A simplified example of the SmartChoices API in C++.

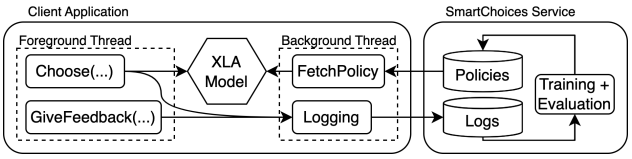

Figure 2. SmartChoices service infrastructure. Policies are trained in a central service and sent to client applications. Inference is local to the client and implemented using XLA.

3 Design Overview

SmartChoices has two major deployment settings: service and in-process. In the service setting, a central service collects data, trains models, and periodically transmits new policies back to the client. In contrast, in the in-process setting, data is collected and models for the policy are trained on the client machine. Both deployment settings provide very low latency, safety and ease of use. We discuss key design decisions that support these properties, with more detail provided for the service setting. Note that the same simple SmartChoices API (shown in Fig. 1) is used in both settings.

3.1 Service SmartChoices

The overall infrastructure for service SmartChoices is shown in Fig. 2.

SmartChoices uses local policies, enabling client-side inference. Model graphs and weights for policies are loaded from the service by the client application and a local model is instantiated by the XLA just-in-time compiler (Leary & Wang, 2017). Client code does not have to wait for compilation to complete before calling `Choose`: the `SmartChoice` object will safely fall back to the `default` arm (see §4.2) until a policy is ready. However, it may block on policy readiness if preferred.

Local policies have several advantages. First, they enable very low decision latency; for the applications in §6, typical median latency is $O(10)$ μs and can be as low as 2 μs. Requests to the central service are reduced. Unit tests and production client code can use the same code paths with-

out complex mock objects. Finally, inference is robust to network issues, since the main application thread performs inference without communication with the SmartChoices service.

Using local policies does limit us to models that can fit in RAM for a single machine. However, in practice, this is sufficient to outperform existing heuristics for a wide variety of applications (see §6).

Communication with the service is asynchronous and handled by separate background threads. Client applications need to communicate with the SmartChoices service to *(i)* send logged data to train and evaluate new policies and *(ii)* receive policy updates. This communication is handled by a background thread initialized when the `SmartChoice` object is instantiated. Logged data generated by `Choose` and `GiveFeedback` calls are buffered in a shared queue by the foreground thread, and then sent in batches by the background thread. Policy updates are obtained by polling the service. As with the use of local policies, asynchronous communication via background threads ensures that `Choose` and `GiveFeedback` calls are low latency and do not block on network issues.

Training and evaluation for all clients uses the same service code. Logged data is sent to the service in a standard log format and contains all information required to train *and* evaluate new policies. These logs are thin wrappers around the Protocol Buffers defining the input, output and feedback types (see Fig. 3). For training, `Choose` logs include x_t, A_t, a_t, and metadata about the policy π, and `GiveFeedback` logs include the metric values y_t and the ID of the corresponding `Choose` call. For evaluation, `Choose` logs also include the default arm and arm selection policies. This enables code to be shared, reducing the risk of errors or behavioral differences due to divergent code.

Common steps in the ML pipeline are automated. Logged data is collected in a centralized database, and the service periodically collects summary statistics on the data. If new data is available, the service automatically begins creating new policies. This process involves feature normalization, training (either incremental or from scratch), model evaluation, computing constraints based on desired system-level behavior for metric-constrained optimization (§6.5), and automated analysis and validation checks (§5.3.1) prior to policy rollout.

In addition, hyperparameter tuning is available on demand. By integrating with an industrial scale black box optimization platform, we perform hundreds of training trials with a single command, trying different architectural and training hyperparameter configurations in order to identify the parameters that minimize loss on the evaluation holdout dataset.

Engineers customize SmartChoices via a single configuration file. Almost all of the steps in our ML pipeline can be customized; for example, engineers can configure how features should be normalized or what validation checks should block a policy from being rolled out to client binaries. Enabling engineers to customize all of this via a single configuration file not only makes using SmartChoices easier but increases the likelihood that unintended changes are caught in code review.

Feedback is flexible. The `feedback_handle` object required to `GiveFeedback` (see Fig. 1) can be restored from an ID, allowing feedback to be provided days later in a separate process. This is particularly useful for engineers who are only able to measure the quality of the decision after some delay. In addition, the `Choose` and `GiveFeedback` calls can be tied together by an ID provided by client code; this often significantly reduces the amount of infrastructure engineers need to add to use SmartChoices (e.g., storing a mapping from a SmartChoices ID to their ID).

3.2 In-Process SmartChoices

Some client applications do not want to depend on a hosted logging and training service. Reasons include strict constraints around privacy or data sovereignty, a need to adapt rapidly to recent data, or a need for even lower latency inference ($< 1\ \mu s$).

SmartChoices supports such applications either directly linking in trained policies in the client binary or training locally (i.e., within the same binary making decisions) in background threads. No data leaves the client binary in either case. Local training enables very rapid adaptation to data: the default latency from when feedback is provided to when the data is trained on is $200\,\mathrm{ms}$. This is configurable, with resultant trade-offs in CPU usage and speed of adaptation.

Local training shares some key design decisions with Service SmartChoices (§3.1). In particular, local training for different SmartChoices applications uses the same code paths, and engineers also customize local training via a single configuration file.

4 SAFETY

4.1 Models as Code

To date, people express most computations in traditional software (as opposed to ML models). This has huge advantages for building, testing, maintaining, and modifying complex systems. However, some desired behaviors – say, recommending good items to an end-user – do not admit a concise description in code but must instead be learned from data. In practice, this nearly always means defining a parametric function class m_θ and then searching for

parameters $\theta \in \Theta$ to optimize some objective(s). This *training* processes bears no resemblance to traditional software engineering via human brains and keystrokes, and is usually decoupled from the typical software release cycle. Still, models are code. In theory they can implement any function (Zhou, 2020), and in practice we see continuous improvement in capabilities as the field progresses.

Over time, engineering best practices have tended to treat ML models more and more like code, e.g., tracking data and experiments and versioning models. SmartChoices builds on this by allowing engineers to rely on their existing integration testing, canary, release, and rollback processes. For example, engineers can choose to link trained models directly into application binaries; in this case, undoing a problematic policy rollout is as simply as rolling back the binary version. Alternatively, engineers can specify environment-dependent policy tags (§5.3.3). This allows policy updates to be first deployed in a staging environment before being used for all production traffic.

4.2 Specifying a Default Action

We require SmartChoices users to provide a *default action* when calling into SmartChoices. This has several advantages. First and foremost, it provides a safe fallback in case of any error. Even with rigorous software engineering practice, bugs will arise: via logical errors, numerical instabilities, or low level compiler bugs. Conducting model inference in-memory on CPU allows us to detect failures (e.g., malformed inputs, infinite predicted rewards, or errors in the XLA compiler) without overhead. If a failure is detected, we fall back to outputting the default action.

Secondly, providing the default allows us to automatically setup a long-running "holdback" experiment, i.e., we choose the default uniformly at random some fraction of the time in order to A/B test our learned implementation against the default.

Even without an explicit holdback, we can use the identity of the default action to estimate the metrics for a policy that always choose the default action (henceforth the "default policy") via counterfactual policy evaluation or CPE (Bottou et al., 2013) .

Finally, having the default allows us to implement imitation learning against the default policy and regularize to it, penalizing deviations from it. This allows us to bootstrap from a baseline policy that achieves the current system performance.

4.3 Fairness

As computational systems take on increasing influence in society, it has become increasingly important to understand the implications, and, ideally, design systems that encourage healthy outcomes for businesses and society at large. This presents a vast research frontier (see e.g., Chouldechova & Roth (2020)) that is currently actively being explored, even at the foundational level of appropriate formal definitions of fairness[3]. Still, whatever best practices around ML fairness emerge over time, treating ML models as code and fundamentally tying models to the decisions they result in has the advantage of providing a centralized surface to design, implement, and monitor compliance with the desired behavioral constraints in production.

4.4 Testing and Production Readiness

As noted, ML can create novel types of technical debt and production risks. By design, SmartChoices mitigates many of these. For example, Breck et al. (2017) suggest a rubric for scoring ML productionization readiness. As shown in Table 1, SmartChoices integrations automatically meet 19 of the 28 specified criteria, and manually meet 5 more (and could be extended to automatically meet them); the remainder are concerned with feature generation, which remain the responsibility of engineers using SmartChoices.

As for the tests marked "manual" in the table, most have supporting analyzes automatically performed and displayed on the SmartChoices frontend (§5.3.2). Field sensitivity charts show how important each feature was to any specific SmartChoices model's performance, supporting the identification of non-beneficial features for later removal, and suggesting if simpler models may be better. The results of hyperparameter tuning on architectural parameters can surface whether simpler models perform better. Automatic CPE against logged data reveals the impact of model staleness.

Another interesting case is *training / serving skew*, in which feature semantics change between training time and inference time. The SmartChoices workflow discourages a common source of skew, namely the use of different code paths in training and inference[4]. Any skew introduced is due to code or configuration changes in the client system, which can and should be tracked and audited via standard production engineering principles. Ultimately, however, detecting semantic changes in features requires human oversight, and SmartChoices facilitates that by tracking feature distributions and surfacing them to engineers on the frontend.

[3] Some definitions are incompatible, with known impossibility results revealing fundamental tradeoffs.

[4] It takes specific extra development work to enable such skew.

Data Tests	
Feature expectations are captured in a schema.	Automatic
All features are beneficial.	Manual**
No feature's cost is too much.	User responsibility
Features adhere to meta-level requirements.	Yes (for select requirements)
The data pipeline has appropriate privacy controls.	Automatic
New features can be added quickly.	Automatic
All input feature code is tested.	User responsibility
Model Tests	
Model specs are reviewed and submitted.	Automatic
Offline and online metrics correlate.	Automatically measured & surfaced.
All hyperparameters have been tuned.	Automatic
The impact of model staleness is known.	Manual**
A simpler model is not better.	Manual**
Model quality is sufficient on important data slices.	Automatic (if configured)
The model is tested for considerations of inclusion.	Manual**
Infrastructure Tests	
Training is reproducible.	Automatic
Model specs are unit tested.	Automatic
The ML pipeline is Integration tested.	Automatic (with proper integration)
Model quality is validated before serving.	Automatic
The model is debuggable.	Yes
Models are canaried before serving.	Automatic (if configured)
Serving models can be rolled back.	Yes
Monitoring Tests	
Dependency changes result in notification.	User responsibility
Data invariants hold for inputs.	Yes (for selected invariants).
Training and serving are not skewed.	User responsibility (mitigated per §4.4)
Models are not too stale.	Automatic (if configured)
Models are numerically stable.	Automatically guarded against.
Computing performance has not regressed.	Manual**
Prediction quality has not regressed.	Automatic

Table 1. Measuring SmartChoices against the ML Test Score Rubric of Breck et al. (2017). Items marked ** could be made automatic in a straightforward manner.

```
message ExampleContextProtocolBuffer {
  int32 category = 1        [(opts)={num_categories: 15}];
  string textual = 2        [(opts)={max_length:5}];
  repeated float vector = 3 [(opts)={shape: 7}];
  string debug = 4          [(opts)={log_only: true}];
}

message ExampleArmProtocolBuffer {
  option (arm_msg_opts) = {max_num_arms: 10};

  int32 category = 1        [(opts)={num_categories: 15}];
  string textual = 2        [(opts)={max_length:5}];
  repeated float vector = 3 [(opts)={shape: 7}];
  string debug = 4          [(opts)={log_only: true}];
}

message ExampleFeedback {
  float reward = 1     [(opts)={maximize_goal{}}];
  float penalty = 2    [(opts)={minimize_goal{}}];
  float aux_metric = 3 [(opts)={log_only: true}];
}
```

Figure 3. An example context, arm, and feedback Protocol Buffer defining an input, output and feedback type. Syntax simplified for brevity.

5 Integrating With SmartChoices

5.1 Problem and Type Specification

Users of SmartChoices begin by expressing their problem in terms of types. Using Protocol Buffers (Google, 2008), users define (input) *Context*, (output) *Arm*, and *Feedback* types. (See examples in Fig. 3). These types, similar to structs in C, encapsulate a collection of multiple datatypes. Each data element inside a Protocol Buffer is described as a field with a type and a name. Users indicate modeling requirements for a field – such as the size of a categorical feature, or whether a reward should be minimized or maximized – via field annotations.

Protocol Buffers have several advantages: they support reflection and easily adding and removing fields, have cross-language support, support compressed serialization, and are widely used. These benefits enabled us to create generic components for converting Protocol Buffers into encoding tensors suitable for ML models and logging training data. This dramatically cuts down on "glue code" and "pipeline jungles." Additionally, because Protocol Buffer annotations allow SmartChoices users to configure parameters in-line with their field datatypes, using Protocol Buffers reduces a major source of "config debt."

5.2 Instrumentation in User Code

To instrument their code, SmartChoices users begin by *(i)* adding a build rule (Bazel, 2023) parameterized by their configuration file as a target dependency and *(ii)* including the SmartChoices library in their code.

As shown in Fig. 1, users then create a SmartChoices instance at the location in code where they want to use a learned implementation. Next, users call `Choose` with a context proto as well as a set of candidate arms and a default arm. `Choose` selects one of the arms based on the policy, and handles logging the context, arms, and choice for training. User code performs some action as a result of the choice, and measures rewards and penalties which are recorded in a feedback proto passed to the `GiveFeedback` method.

5.3 Service SmartChoices: Additional Tools

Users of the SmartChoices service deployment have access to tools that eliminating the need for custom code to analyze, monitor, and manage machine learning pipelines.

5.3.1 Analysis and Validation

An automated analysis is run using a held-out dataset for every newly-trained policy. Available analysis includes: *(i)* the distribution of chosen arms; *(ii)* per-metric distributions of critic model predictions; *(iii)* per-metric estimates of feature importance; *(iv)* per-metric CPEs comparing the newly-trained policy to three baselines (the default policy, a random policy, and the current "live" policy); *(v)* (for problems with binary feedback) the area under the receiver operating characteristic curve (henceforth, "ROC AUC") and the precision-recall curve; *(vi)* (for Pareto Frontier Search (§2.3.2)) an estimate of the Pareto-optimal tradeoffs achievable by the trained policy.

A policy is *validated* if analysis demonstrates that it has "good" behavior; users can what this means by modifying

their configuration file. By default, a policy is validated if we have at least 95% confidence that it outperforms each of the three baselines, with uncertainty estimated via Poisson bootstraps (Chamandy et al., 2012) of the held-out dataset. Additional validation checks include conditions on just the newly-trained policy (e.g. a minimum ROC AUC) or conditions comparing the newly-trained policy to the current "live" policy (e.g. upper bounds on the statistical distance between the distribution of chosen arms or critic model predictions).

5.3.2 *Monitoring Behavior and Performance*

An automatically generated web frontend shows: *(i)* training progress; *(ii)* the logged distribution of each context, arm, and feedback field over time; *(iii)* the logged reward over time for the trained policy, a random policy, and the "default" policy; *(iv)* the results of automated analysis (§5.3.1) for any policy.

5.3.3 *Managing Policy Rollouts*

Policy rollouts in SmartChoices are managed through policy tags. Each "tag" is a human-readable string referencing a single policy, with the referenced policy updated over time.

Two tags are available by default. The "latest" tag always references the most recently trained policy, while the "live" tag references the most recent trained policy that was validated. Users typically use the "live" policy, but custom rollout (and rollback) strategies can be implemented via custom tags. Client binaries periodically poll the service for updated policies and tag references, and seamlessly switch over to the new policies.

Policy tags also simplify exploring tradeoffs when conducting multimetric optimization (§2.3). Tags are created corresponding to a range of behavior (e.g., ranging from "treat the first metric as 5 times as important" to "treat the second metric as 5 times as important"), and policies for each tag are generated using a holdback evaluation dataset. Users can directly reference these tags in code.

6 Case Studies

SmartChoices has been successfully applied in a diverse range of problem domains, ranging from low-level optimizations (e.g., §6.1) to user-facing applications (e.g., §6.6). We select a representative sample that motivates each of the capabilities introduced in §2 and demonstrates how SmartChoices' design (§3) enables engineers to successfully apply ML in their systems.

Client using existing capabilities (e.g., §6.4) have integrated SmartChoices with $O(days)$ engineering time. Additional engineering work can be required to make features available at `Choose` time (see Fig. 1) or measure the quality of the choice.

6.1 Learned Cache Eviction

SmartChoices reduced the fraction of user-requested bytes missed in a Content Delivery Network (CDN) cache for a large-scale video service by improving its eviction policy. This reduced latency for end users, improved several quality of experience (QoE) metrics and lowered the cost of content distribution.

The learned eviction policy first uses a heuristic to select 4 promising candidates, and then uses SmartChoices to run a 2-stage tournament selection to pick the item to evict. Many heuristics are suitable, but in practice we found selecting the 4 least recently used (LRU) items works well. Feedback is binary: 1 if the next access time for the SmartChoices-selected entry was the latest among the candidates it was compared with, and 0 otherwise.

Two SmartChoices features were key to a successful launch: its low latency (§3) enabled timely *individual* cache eviction decisions, and efficient in-process model training (§3.2) eliminated the need for transfer training data between edge servers and data centers while imposing only minimal compute overhead. In addition, using SmartChoices on top of a decent heuristic (LRU) provided additional production safety, guaranteeing acceptable performance during initial training and later adaptation to new usage patterns. Using SmartChoices decreased the portion of user-request bytes missed by 9.1% at peak traffic, representing a significant improvement over highly-tuned code.

6.2 Optimizing Compilation

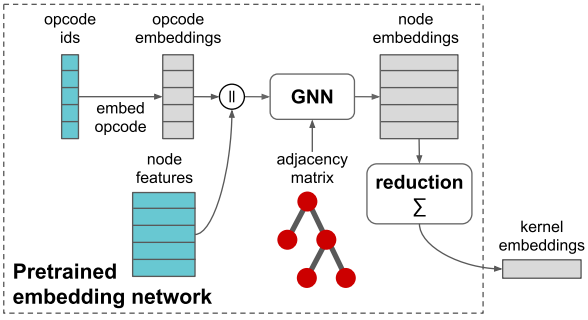

Figure 4. Kernel embedding consumed as input by SmartChoices.

SmartChoices achieved performance gains in a compiler by enabling compiler parameters to be dynamically adjusted, e.g., in response to changes to other components.

ML workloads consume enormous amounts of compute in large industrial settings. Specialized accelerators such as Tensor Processing Units (TPUs) are increasingly used for model training and inference. The XLA compiler (Leary

& Wang, 2017) generates code that can run on these accelerators; optimizing the compiler can improve latency, throughput and cost. *Tile size selection* (Rivera & Tseng, 1999) is one of the most performance-critical optimizations in the XLA TPU compiler, affecting how tensors are moved between distinct levels of the memory hierarchy. The goal is to select an optimal tile size for a high-level operation (HLO) such that its input and output tensors fit in the scratchpad memory, while minimizing its execution time.

Concretely, we must choose from a finite set of arms $A_t \subset \mathbb{R}^d_{\geq 0}$ for operation t, where d is an upper bound on the sum of tensor ranks of all inputs to an operation we wish to consider for optimization.

A naive approach is exhaustive search over all tile sizes. However, since each HLO has a large number of valid tile sizes, and the optimal tile size frequently changes due to active development of the XLA compiler, exhaustive search is intractable.

Existing search-based techniques such as TVM (Chen et al., 2018) cannot be deployed in the XLA compiler since they assume that optimization decision can be made independently from the rest of the graph and require optimizations to be applied at the same stage in the compilation flow. (Phothilimthana et al., 2021)

Instead, using arm features (§2.1), SmartChoices is used to filter out 99% of candidates, with an exhaustive search applied to the remaining 1%. We begin by pretraining a learned cost model (Phothilimthana et al., 2019) that predicts the TPU runtime of an HLO using an historic dataset of actual runtimes. To keep up with changes to the compiler, we fine-tune the model periodically, freezing the graph-embedding network (Fig. 4) and retraining only the feed-forward head. Continuous training (§3.1) for the feed-forward head allow us to stay up to date while avoiding the full cost of end-to-end training. Searching over the candidates selected by SmartChoices achieves $90 - 95\%$ of the speedup achieved by full exhaustive search, scales to optimize all relevant HLOs (vs. 5% coverage for the prior exhaustive search), and is about 29 times faster overall.

6.3 Optimizing Thread Counts

SmartChoices reduced tail latency for end-user queries on a flight booking search service by optimizing thread count.

When processing an end-user query, the service first determines all relevant sequences of flights, or "itineraries". The service then retrieves fares for all subsequences of each itinerary. As itineraries can largely be processed independently, this work can be parallelized across multiple threads.

A fixed thread count of four resulted in reasonable system-wide performance. However, experimental data demonstrated that complex end-user queries benefited significantly from more threads.

For each query, SmartChoices dynamically selected the thread count based on context including the number of flight sub-sequences and the source and destination regions. We then measured latency and CPU usage for that query, providing both as feedback. The resulting contextual bandit policy selects the thread count that minimizes a weighted linear combination of both measures.

SmartChoices' support for mixed-type context (§2.1) via automatic conversion of Protocol Buffers into encoding tensors (§5.1) enabled rapid experimentation with different context features. At launch, average latency reduced by 25% and P99 latency reduced by 16% without a significant increase in CPU cost.

6.4 Optimized Work Partitioning

SmartChoices improved data availability and freshness for a service that monitors machine learning workloads via dynamic work rebalancing (Kirpichov & Denielou, 2016) between monitoring tasks.

Each monitoring task summarizes telemetry (e.g., RAM and accelerator usage) for a list of workloads. Tasks that are too large will take a long time to complete, resulting in stale or missing data. In contrast, tasks that are too small incur unnecessary overhead, and can result in outages if the *monitoring* workload exceeds its allocated capacity.

When a monitoring task begins running, it can either `shard` (scheduling two smaller tasks covering the relevant workloads) or `execute`. SmartChoices optimizes this decision. The reward for `shard` is always zero, while the reward for `execute` is the difference $t_d - t_e$ between the target maximum data staleness t_d and the actual execution time t_e.

This approach encourages sharding only tasks for which the expected execution time is greater than the deadline t_d. The reward formulation allows feedback to be provided to the model immediately for continuous re-training. Context for each decision includes information about the data source being queried, the number of entities contained in the shard and the time and day of the week.

As in §6.3, SmartChoices' support for mixed-type context enabled rapid experimentation with context features. In addition, automated training and validation (§3.1) enables policies adapting to changes in query distributions (e.g., from a sudden increase in long-running queries) to be deployed daily.

Using SmartChoices significantly reduced alerts on missing or stale data, translating directly into time savings for the team responsible for maintaining the service. In addition, 53% fewer tasks hit the execution deadline.

6.5 Optimizing Refresh Rates

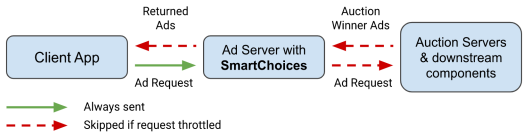

Figure 5. Use of SmartChoices in the ads service. The app continues to send ad requests as before. Throttling requests in the ad server allows us to skip subsequent steps, including the computationally-costly auction.

SmartChoices optimized resource usage on ad refreshes for an ads service supporting a widely used product. Ads viewed by users are stored locally within an app on their mobile devices, and these ads are periodically refreshed via the process shown in Fig. 5 so that users can view new ads. The computational resources used to run the ad auction to fulfil the refresh request are considered to be wasted if the refreshed ad is not viewed.

The Ads team used SmartChoices to throttle those ad requests from the app that are less likely to result in an ad view. SmartChoices uses constrained optimization (§2.3.1) to do so. Feedback for training is binary: 1 if the ad is viewed and 0 otherwise.

The SmartChoices policy involves a critic model predicting the probability that refreshing the ads will result in an ad view (`pView`), with requests with `pView` below a threshold q throttled. q controls the tradeoff between resource usage and the number of ad views (and downstream metrics such as clicks). Increasing q saves more resources at the potential cost of a reduction in views. In practice, we found that different tradeoffs are appropriate for different device types; as such, we use a single critic model to predict `pView` but specify $q_0, q_1, ...$ per device type.

Instead of selecting a fixed threshold, the Ads team wanted to ensure that SmartChoices throttled approximately the same *fraction* of traffic over time. We eliminated the need for manual tuning when training a new model by automatically computing the appropriate `pView` threshold based on the *distribution* of that model's predictions on a holdout dataset.

To guarantee that policy updates do not drastically impact downstream systems, we compare throttling decisions of the "live" policy with new policies during validation (§5.3.1). New policies are validated for use only if the change in throttling decisions is minimal for all device types.

SmartChoices was deployed in three stages. Overall, ad requests to ad servers were reduced by 5.8% while views increased by 6.3%.

1. Phase 1 reduced in ad requests to ad servers of 12%, with no change in views and downstream metrics.
2. Phase 2 removed a heuristic filter on inactive users. This increased views by 1.4% with only a 4% increase in ad requests.
3. Phase 3 doubled the frequency of ad requests from the client app while adjusting the thresholds on `pView`. This increased views by 4.8% with only a 2.2% increase in ad requests.

6.6 Optimizing User Experience

SmartChoices improved user-engagement metrics across a variety of User Experience (UX) optimization applications by identifying the best of a set of human-designed candidate options. Example applications include:

- Selecting the best notification string to inform a user that their storage space is running low or has been exhausted.
- Selecting the best notification string to inform a user about new personalized curated content for them.
- Selecting the best string during mobile device onboarding, to guide the user to enable features of interest.

For these applications, using SmartChoices yielded a 2% to 10% improvement in user-engagement metrics.

Prior to SmartChoices, the typical solution for such UX optimization problems was A/B testing. This approach has several disadvantages:

- **Poor scalability**: Setting up and running each A/B experiment is a manual process, typically requiring significant engineering effort. As such, the number of such live experiments teams run is usually much smaller than the number of potential applications they want to optimize.
- **Limited personalization and contextualization**: The candidate that performs best overall may not be the best for a specific sub-population of users or in specific scenarios (for example, if the same user is using a different device). Optimizing for each contextual feature potentially requires a separate A/B experiment.
- **Non-adaptive**: After conducting one A/B experiment, the teams usually do not know whether the best-performing candidate has changed unless they conduct another A/B experiment.

SmartChoices addresses all of these issues. Continuous model re-training, validation and deployment (§3.1) enables

SmartChoices to adapt to external changes automatically. With contextual bandits at its core (§2.1), SmartChoices takes advantage of contextual and/or user features to optimize for specific users. Simple APIs (see Fig. 1) abstract away cumbersome machine learning workflows, enabling SmartChoices to be easily integrated with multiple applications.

Indeed, these advantages over traditional A/B testing were the primary reason the teams adopted SmartChoices. Although initial adoption usually entails some upfront effort, teams are usually able to easily expand SmartChoices to related applications after the initial integration.

6.7 Ranking Recommendation Cards

SmartChoices increased the rate at which business owners completed tasks by optimizing the order in which recommendation cards are displayed.

Each card prompts business owners (here, "users") to complete a different action (e.g., upload phone numbers, respond to custom reviews). Our goal is to show the most relevant cards to users. Ordering affects visibility; only the first three items are visible without scrolling.

Prior to using SmartChoices, cards were simply ordered by their global click-through rates (CTR). SmartChoices was deployed in two phases:

1. Phase 1 used click-based feedback, with a reward of 1 when end-users click on a card and 0 otherwise. We used this feedback since we expected optimizing CTR *contextually* to improve overall CTRs. Other important metrics also saw significant improvements: $\mathbf{8.7}\%$ more users interacted with cards, and the count of 28-day active users increased $\mathbf{27}\%$.

2. Phase 2 used task completion feedback, with a reward only when end-users completed the task for a card. This problem framing allowed us to focus on the most important tasks by appropriately scaling rewards. For example, $\mathbf{0.8}\%$ more users updated their business profile, while $\mathbf{2.2}\%$ more users visited a page summarizing for their profile. This represented a significant increase on a large user base. While CTR decreased, we observed no change to the overall task completion rate, indicating a reduction in low-task-completion-intent clicks.

Once again, automated training and validation (§3.1) enables policies to to adapt to seasonal changes automatically; for example, the "Holiday Hour Edits" card is automatically ranked higher before local holidays.

	R	L	DS	SC
Ranking		✓		✓(§2.2)
Pareto Frontier Search		✓		✓(§2.3.2)
Constrained Optimization				✓(§2.3.1)
Evaluation via CPE	✓	✓	✓	✓(§5.3.1)
Software-centric API		✓		✓(§5)
Resilient to Service Outage			✓	✓(§3.1)
Local Low-Latency Inference				✓(§3.1)

Table 2. A comparison of SmartChoices against other projects deploying ML for decision making within production systems on key features. *R: ReAgent, L: Looper, DS: Decision Service, SC: SmartChoices.*

7 Related Work

Carbune et al. (2019) presented an earlier prototype of SmartChoices. The version we present here has some significant differences based on our production experience, most notably a focus on contextual bandits over general reinforcement learning[5], assorted safety features, and a service architecture. Abadi & Plotkin (2021) explored how programming languages can be endowed with operational and denotational semantics corresponding to learned decision policies (including stochastic ones), and prove these semantics coincide.

Several other projects have explored related aspects of how to effectively deploy ML for decision making within production systems. We summarize key differences in Table 2, with more detail provided below.

ReAgent (Gauci et al., 2018) (previously known as Horizon) is a platform for reinforcement learning in production with similar goals as SmartChoices but different feature sets. SmartChoices differs on ease of use (e.g., providing more automation for data preprocessing, training, and model updates), modeling emphasis (e.g., multimetric optimization [§2.3]), and scope (e.g., suitability for low–level applications like §6.1).

Looper (Markov et al., 2022) is an end-to-end platform for training and deploying machine learning models, with particular emphasis on optimizing product goals with parameterized decision functions which the authors call "smart strategies," using a combination of immediate observations (for each decision) and product metrics (measured from the aggregate effect of a large set of decisions). In effect, it trains models for all observed metrics, then optimizes over a class of parameterized "strategy blueprints" with respect to long-term product metrics using a sequence of A/B tests selected via Bayesian optimization. Unlike Looper, SmartChoices is not tightly integrated with an A/B experiment framework and doesn't have a direct notion of product goals[6]. Additionally, Looper must be called via RPC, and

[5]To be clear, our design accommodates RL quite naturally.

[6]In practice, a grid search over SmartChoices scalarization

requires users to implement fallback logic in case of failure, in contrast to our default action. Finally, Looper's median inference latency is reported at $2\,\mathrm{ms}$, and feature extraction latency at $45\,\mathrm{ms}$ – about 3 orders of magnitude slower than SmartChoices – preclude its use in many systems applications.

Microsoft's Decision Service (Agarwal et al., 2016) provides a service for contextual bandits, and shares many design goals with SmartChoices. As a cloud service, it has hosted logging, training, and inference, however it also supports loading models into the client for faster local decision making. SmartChoices has considerably lower decision latency, however, making it suitable for additional low level system optimizations (as low as $O(1)$ μs vs. a reported average latency of $0.2\,\mathrm{ms}$ in (Agarwal et al., 2016)), as well as capabilities beyond standard contextual bandits (c.f., §2).

Natarajan et al. (2020) investigate *programming by rewards* (PBR), whereby system performance can be used to aid the programming process – either by filling in values from a user-provided template (i.e., programs with missing constants), or by generating programs within a limited class representable by fixed–depth decision trees. Like SmartChoices, PBR searches for a reward maximizing implementation, however it is restricted to learn functions of type $\mathbb{R}^m \to \mathbb{R}^n$, owing to the type of training used. It has a different deployment model in which learned implementations are translated directly into source code that is checked-in. This has advantages in terms of interpretability, speed, and avoiding additional dependencies. However, only a restricted of class of implementations are considered, and there are no affordances for adapting implementations to changing environments over time.

Two classes of problems similar to contextual bandits are bayesian optimization (BO) and reinforcement learning (RL). BO focuses on the low-data regime where gathering feedback is expensive; in contrast, we focus on settings with more abundant data. In the RL setting, the arm selected at time t affects the next context x_{t+1}. Since a large class of practical optimizations can be framed as a contextual bandit problem, we have not yet needed to support RL.

Contextual bandit models have been used across industry to solve a wide range problems. For example, contextual bandits can replace the typical A/B testing process for deciding on UI changes. Instead of manually comparing the relative performance of two fixed UIs, contextual bandits can learn to personalize UI elements to maximize each user's experience. Prominent examples include deciding the relative position of news stories (Li et al., 2010), selecting thumbnail artwork for video content (Amat et al., 2018), and ordering products on a "carousel" (Ermis et al., 2020). Industrial applications also use contextual bandit models in the backend to solve problems in dynamic or ambiguous environments. These use cases include disambiguating ambiguous verbal requests of smart speakers (Moerchen et al., 2020), personalizing the recommendations of products (Sawant et al., 2018), and determining user "intent" when interacting with support chat bots (Sajeev et al., 2021). The wide range of applications speaks to the huge potential impact of an approach like SmartChoices that accelerates improving systems with ML.

parameters has sufficed for many applications. Also, there's no fundamental obstacle to automating this parameter search.

8 Acknowledgements

This work benefited from the wisdom of many colleagues engaged in the development of machine learning in mature production systems. We especially wish to thank Jeff Dean, Sanjay Ghemawat, Jay Yagnik, Andrew Bunner, George Baggott, Jesse Berent, Ben Solnik, Alex Grubb, Weikang Zhou, Eugene Kirpichov, Arkady Epshteyn, Ketan Mandke, Wei Huang, and Eugene Brevdo for thoughtful input into the design and goals of the SmartChoices project. Lastly, we would like to thank our many colleagues who integrated SmartChoices into their projects and championed this effort within their respective teams.

References

Abadi, M. and Plotkin, G. Smart choices and the selection monad. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pp. 1–14. IEEE, 2021.

Agarwal, A., Bird, S., Cozowicz, M., Hoang, L., Langford, J., Lee, S., Li, J., Melamed, D., Oshri, G., Ribas, O., et al. Making contextual decisions with low technical debt. *arXiv preprint arXiv:1606.03966*, 2016.

Agrawal, S., Devanur, N. R., and Li, L. An efficient algorithm for contextual bandits with knapsacks, and an extension to concave objectives. In *Conference on Learning Theory*, pp. 4–18. PMLR, 2016.

Amat, F., Chandrashekar, A., Jebara, T., and Basilico, J. Artwork personalization at Netflix. In *Proceedings of the 12th ACM conference on recommender systems*, pp. 487–488, 2018.

Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., Koo, C. Y., Lew, L., Mewald, C., Modi, A. N., Polyzotis, N., Ramesh, S., Roy, S., Whang, S. E., Wicke, M., Wilkiewicz, J., Zhang, X., and Zinkevich, M. TFX: a TensorFlow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1387–1395, 2017.

Bazel. Rules — bazel, Feb 2023. URL https://bazel.build/extending/rules.

Bottou, L., Peters, J., Quiñonero-Candela, J., Charles, D. X., Chickering, D. M., Portugaly, E., Ray, D., Simard, P., and Snelson, E. Counterfactual reasoning and learning systems: The example of computational advertising. *Journal of Machine Learning Research*, 14(11), 2013.

Breck, E., Cai, S., Nielsen, E., Salib, M., and Sculley, D. The ML test score: A rubric for ML production readiness and technical debt reduction. In *Proceedings of IEEE Big Data*, 2017.

Carbune, V., Coppey, T., Daryin, A., Deselaers, T., Sarda, N., and Yagnik, J. SmartChoices: hybridizing programming and machine learning. *ICML 2019 Workshop on Reinforcement Learning for Real Life*, 2019.

Chamandy, N., Muralidharan, O., Najmi, A., and Naidu, S. Estimating uncertainty for massive data streams. Technical report, Google, 2012.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., et al. Tvm: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

Chouldechova, A. and Roth, A. A snapshot of the frontiers of fairness in machine learning. *Communications of the ACM*, 63(5):82–89, 2020.

Ermis, B., Ernst, P., Stein, Y., and Zappella, G. Learning to rank in the position based model with bandit feedback. In *CIKM 2020*, 2020.

Gauci, J., Conti, E., Liang, Y., Virochsiri, K., He, Y., Kaden, Z., Narayanan, V., Ye, X., Chen, Z., and Fujimoto, S. Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260*, 2018.

Gelbart, M., Snoek, J., and Adams, R. Bayesian optimization with unknown constraints. In Zhang, N. and Tian, J. (eds.), *Uncertainty in Artificial Intelligence - Proceedings of the 30th Conference, UAI 2014*, pp. 250–259, 2014.

Google. Protocol buffers: Google's data interchange format, 2008. URL https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html.

Grover, A., Wang, E., Zweig, A., and Ermon, S. Stochastic optimization of sorting networks via continuous relaxations. In *ICLR*, 2019.

Kirpichov, E. and Denielou, M. No shard left behind: dynamic work rebalancing in google cloud dataflow, 2016. URL https://cloud.google.com/blog/products/gcp/no-shard-left-behind-dynamic-work-rebalancing-in-

Leary, C. and Wang, T. XLA – tensorflow, compiled, 2017. URL https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

Li, L., Chu, W., Langford, J., and Schapire, R. E. A contextual-bandit approach to personalized news article recommendation. In Rappa, M., Jones, P., Freire, J., and Chakrabarti, S. (eds.), *Proceedings of the Nineteenth International Conference on the World Wide Web (WWW 2010)*, pp. 661–670. ACM, 2010. ISBN 978-1-60558-799-8. URL http://www.research.rutgers.edu/~lihong/pub/Li10Contextual.pdf.

Markov, I. L., Wang, H., Kasturi, N. S., Singh, S., Garrard, M. R., Huang, Y., Yuen, S. W. C., Tran, S., Wang, Z., Glotov, I., Gupta, T., Chen, P., Huang, B., Xie, X., Belkin, M., Uryasev, S., Howie, S., Bakshy, E., and Zhou, N. Looper: An end-to-end ml platform for product decisions. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, pp. 3513–3523, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393850. doi: 10.1145/3534678.3539059. URL https://doi.org/10.1145/3534678.3539059.

Moerchen, F., Ernst, P., and Zappella, G. Personalizing natural-language understanding using multi-armed bandits and implicit feedback. In *CIKM 2020*, 2020.

Natarajan, N., Karthikeyan, A., Jain, P., Radicek, I., Rajamani, S., Gulwani, S., and Gehrke, J. Programming by rewards. *arXiv preprint arXiv:2007.06835*, 2020.

Phothilimthana, M., Burrows, M., and Kaufman, S. J. Learned TPU cost model for XLA tensor programs. In *Workshop on ML for Systems at NeurIPS*, 2019.

Phothilimthana, P. M., Sabne, A., Sarda, N., Murthy, K. S., Zhou, Y., Angermueller, C., Burrows, M., Roy, S., Mandke, K., Farahani, R., et al. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 1–16. IEEE, 2021.

Rivera, G. and Tseng, C.-W. A comparison of compiler tiling algorithms. In *International Conference on Compiler Construction*, pp. 168–182. Springer, 1999.

Sajeev, S., Huang, J., Karampatziakis, N., Hall, M., Kochman, S., and Chen, W. Contextual bandit applications in a customer support bot. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 3522–3530, 2021.

Sawant, N., Namballa, C. B., Sadagopan, N., and Nassif, H. Contextual multi-armed bandits for causal marketing. In *ICML 2018*, 2018.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015.

Sun, W., Dey, D., and Kapoor, A. Safety-aware algorithms for adversarial contextual bandit. In *International Conference on Machine Learning*, pp. 3280–3288. PMLR, 2017.

Zhang, R. and Golovin, D. Random hypervolume scalarizations for provable multi-objective black box optimization. In *International Conference on Machine Learning*, pp. 11096–11105. PMLR, 2020.

Zhou, D.-X. Universality of deep convolutional neural networks. *Applied and computational harmonic analysis*, 48 (2):787–794, 2020.