# SmartChoices: Augmenting Software with Learned Implementations

**Daniel Golovin**[*] **Gábor Bartók**[†] **Eric Chen**[†] **Emily Donahue**[*] **Tzu-Kuo Huang**[*]
**Efi Kokiopoulou**[†] **Ruoyan Qin**[*] **Nikhil Sarda**[†] **Justin Sybrandt**[*] **Vincent Tjeng**[*]
[*]Google DeepMind [†]Google Research
`smartchoices-2023-authors@google.com`

## Abstract

We are living in a golden age of machine learning. Powerful models perform many tasks far better than is possible using traditional software engineering approaches alone. However, developing and deploying these models in existing software systems remains challenging. In this paper, we present SmartChoices, a novel approach to incorporating machine learning into mature software stacks easily, safely, and effectively. We highlight key design decisions and present case studies applying SmartChoices within a range of 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. However, training and deploying these models in production systems is often fraught with new failure modes and distinct forms of technical debt [25].

This paper re-envisions the workflows to deploy machine learning in large-scale production systems, with an eye towards significantly reducing engineering effort and scope for errors. The result, *SmartChoices*, uses ML models to provide *learned implementations* within an application. We tie models directly to application behavior and eliminate separate development workflows for pipeline management.[1] SmartChoices' design occupies 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. A simple user interface, combined with a design guaranteeing low latency, reliability & safety, enables SmartChoices users to successfully deploy ML to address a diverse range of systems problems.

## 2 Scope and Capabilities

Machine learning can be applied towards many different problem classes. SmartChoices learns implementations for contextual bandits problems; this covers many applications (see §A). Concretely, SmartChoices addresses the following problem class. A system is faced with a sequence of decisions. 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). $A_t$ is a subset of the universe of arms $\mathcal{A}$. The system chooses an arm $a_t \in A_t$ and receives feedback $y_t \in \mathbb{R}^k$ indicating arm quality with respect to $k \geq 1$ metrics. The goal is to provide an implementation via a *policy* $\pi : \mathcal{X} \times 2^{\mathcal{A}} \to \mathcal{A}$ to optimize the metrics[2]; SmartChoices generates policies by post-processing a learned critic model $m : \mathcal{X} \times \mathcal{A} \to \mathbb{R}^k$ targeting the metrics.

[1]Notably, this diverges considerably from the design philosophies of ML platforms that enable setting up arbitrary ML pipelines, such as TFX [5] & Kubeflow [15].

[2]We will refer to metrics we wish to maximize as *rewards* and those we wish to minimize as *costs*.

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

- **Time-varying arm sets**: Require $a_t \in A_t$ when $A_t \subset \mathcal{A}$, via selection masks in the policy.
- **Mixed-type contexts**: Numeric, enumeration, and string contexts are supported via automatically-generated embedding layers in the critic model for $\pi$.
- **Arm-features**: Describing each arm with mixed-type features (henceforth arm-features) is supported via embedding layers (as with contexts). This enables generalization across arms.
- **Multimetric optimization**: There are often several metrics of interest, with inevitable tradeoffs. SmartChoices supports two variants of multimetric optimization, based on metric–constraints and scalarization respectively. See Appendix C for more details.

## 3 User Interface

SmartChoices' user interface is designed to be simple and foolproof. Deploying a learned policy simply requires *(i)* specifying the problem, with any additional configuration options in a single file; *(ii)* adding <10 additional lines of code to the application (see Fig. 2).

### 3.1 Problem Specification

SmartChoices users specify their problem in terms of *Context*, *Arm*, and *Feedback* types. (See Fig. 1). Each type is defined as a protocol buffer (henceforth *protobuf*) message [13]; data elements within messages are described as a field with a type and a name. Users indicate field-specific modeling information (e.g., the size of a categorical feature, whether a field should be ignored for modelling purposes, whether a feedback field should be maximized / minimized) via field annotations.

```
message ExampleContext {
  int32 category = 1        [(opts)={num_categories: 23}];
  string name = 2           [(opts)={max_length: 5}];
  repeated float dims = 3   [(opts)={shape: 2}];
  string debug_info = 4     [(opts)={log_only: true}];     // not used by critic model
}
```

Figure 1: Example context type. Syntax simplified for brevity.

Having a clear separation between the structured representations users work with (*i.e.,* protobufs) and the underlying data representation ingested by the model allows us to surface better errors (e.g., "this named field is unset") and more informative diagnostics (e.g., per-field summary statistics).

Using protobufs specifically provides several additional benefits. **Support for reflection** makes it simple to create generic components for converting any protobuf object into encoding tensors suitable for training and deploying ML models. These components eliminate the need for "glue code" and "pipeline jungles" [25] for feature extraction in user code. **Field annotations** allow users to configure parameters in-line with their field datatypes, reducing risk of "configuration debt" [25]. **Cross-language generated code** for protobuf objects enables turnkey cross-language SmartChoices support. **Compressed serialization** makes logging data memory- and bandwidth- efficient.

In addition to field-specific modelling information, SmartChoices users can configure additional settings for their learned policy. These settings include access controls to diagnostic information, how new policies are trained, evaluated, and monitored, and so on. All of these settings are specified in a single configuration file checked into a repository. This makes it simple to compare the difference between two configurations and enforce that configuration changes undergo a full code review [25].

### 3.2 Using the Learned Policy

Having defined the problem, SmartChoices users instrument their code in two steps. First, users need to ensure that the built application has access to the compiled configuration file. In Bazel [6], this is done by declaring a dependency on a build rule in the application's build target. Users then call the SmartChoices API in code (see Fig. 2); no separate user code is needed for logging data for training.

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

ExampleContext context =          // protobuf with context features; see Fig. 1.
  ExampleContextBuilder().SetCategory(3).SetName("foo").SetDims({2, 5}).SetDebugInfo("bar");
ExampleArm selected_arm;
// (i) note explicit required `default_arm`.
auto feedback_handle = smartchoice.Choose(context, default_arm, candidate_arms, &selected_arm);
// [... user performs action with `selected_arm` ...]
ExampleFeedback feedback = ...;  // protobuf with feedback measuring impact of action
feedback_handle.GiveFeedback(feedback);
```

Figure 2: An abbreviated example of calling the SmartChoices API in C++.

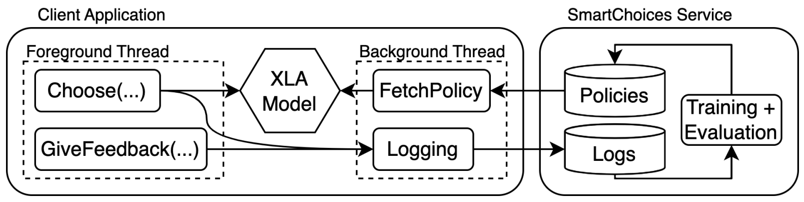

Figure 3: SmartChoices service infrastructure. Policies are trained in a central service and sent to the client application. Inference is local to the client application and implemented using XLA.

We highlight three key features of the API:

**A default arm is required.** This provides three advantages. First and foremost, it provides a safe fallback in case of any detected errors. Next, we can use the identity of the default arm to estimate the metrics for a policy that always chooses the default arm (henceforth the "default policy") via counterfactual policy evaluation (CPE) [7] . Finally, having the default arm allows us to implement imitation learning against the default policy and regularize to it by penalizing deviations from it. This allows us to bootstrap from a policy that achieves the current system performance.

**`selected_arm` is provided directly.** Eliminating user post-processing of model output is not only convenient for SmartChoices users but ensures that all data required for training and analysis (e.g., $A_t$, metadata about policy $\pi$) is available at `Choose` and `GiveFeedback` time.

**Feedback handles are restorable.** The `feedback_handle` object required to `GiveFeedback` can be restored from an ID, allowing feedback to be provided days later in a separate process. This is particularly useful for users who are only able to measure the quality of the decision after some delay. In addition, `Choose` and `GiveFeedback` calls can optionally be tied together by an ID provided by user code, significantly reducing the amount of infrastructure users need to add to use SmartChoices (e.g., storing a mapping from a SmartChoices ID to their ID).

## 4 Implementation

Many different implementations of the SmartChoices interface are possible. Our goals are to: *(i)* provide low-latency inference; *(ii)* guarantee reliability and safety; *(iii)* eliminate the need for custom code to analyze, monitor and manage deployments. We have two implementations: *in-process* (see Appendix D) and *service* (see Fig. 3). These design principles underpin the service implementation:

**Inference is local.** Policies are backed by local models running on CPU. This means that `Choose` is non-blocking and has median latency of $\sim$10 µs. Local inference is also key to our reliability and safety guarantees. First, a trained policy is available even if the network is temporarily down. In addition, bugs causing inference errors can be identified at test time via standard unit tests employing test models, and any inference errors at runtime can be detected immediately, allowing graceful fallbacks to the user-specified default arm. Client applications can seamlessly switch to new policies since models are instantiated by the XLA just-in-time (JIT) compiler [16]. Local inference *does* limit SmartChoices users to models that can fit in a single machine's RAM, but this suffices in practice to outperform existing heuristics for a wide variety of applications (see Appendix A).

**All communication is asynchronous.** `Choose` and `GiveFeedback` calls write training data to a shared queue. Background threads initialized when the `SmartChoice` object is instantiated periodi-

| **Data Tests** | | **Infrastructure Tests** | |
|---|---|---|---|
| Feature expectations are captured in a schema. | ✓ | Training is reproducible. | ✓ |
| All features are beneficial. | ✎ | Model specs are unit tested. | ✓ |
| No feature's cost is too much. | ✋ | The ML pipeline is integration tested. | ✓ |
| Features adhere to meta-level requirements. | ✓ | Model quality is validated before serving. | ✓ |
| The data pipeline has appropriate privacy controls. | ✓ | The model is debuggable. | ✓ |
| New features can be added quickly. | ✓ | Models are canaried before serving. | ✓🔧 |
| All input feature code is tested. | ✋ | Serving models can be rolled back. | ✓ |
| **Model Tests** | | **Monitoring Tests** | |
| Model specs are reviewed and submitted. | ✓ | Dependency changes result in notification. | ✋ |
| Offline and online metrics correlate. | ✓ | Data invariants hold for inputs. | ✓ |
| All hyperparameters have been tuned. | ✓ | Training and serving are not skewed. | ✋🛡 |
| The impact of model staleness is known. | ✎ | Models are not too stale. | ✓🔧 |
| A simpler model is not better. | ✎ | Models are numerically stable. | ✓ |
| Model quality is sufficient on important data slices. | ✓ | Computing performance has not regressed. | ✎ |
| The model is tested for considerations of inclusion. | ✎ | Prediction quality has not regressed. | ✓ |

| | |
|---|---|
| ✓ | met by design (🔧 indicates configurable behavior) |
| ✎ | users must handle manually based on analysis automatically performed and displayed on our frontend (§4) |
| ✋ | user responsibility (🛡 indicates using SmartChoices simplifies passing this test) |

Table 1: Measuring SmartChoices against the ML Test Score Rubric of Breck et al. [8]. SmartChoices deployments meet most tests by design. Tests that remain user responsibility all involve feature generation. SmartChoices reduces the likelihood of training/serving skew (🛡) by *(i)* discouraging use of different code paths in training and inference, and *(ii)* simplifying detection of changes.

cally retrieve the latest validated policy and transmit batches of training data to the central service by reading from the queue. (Note: before a model is ready, calls to `Choose` return the default arm.)

**Users rely on the same infrastructure.** SmartChoices' use of protobufs (§3.1) enables data for all SmartChoices users to be serialized in the same logging format. In addition, compatible ML models are constructed directly from protobuf definitions. This enables a single well-tested code path to be used for preprocessing (e.g., normalizing model input), model training, and analysis.

**Analysis is automated.** For every new SmartChoices policy, the service automatically performs a suite of evaluations using a hold-out dataset. These include feature importance & prediction distributions for the critic model, selected arm frequencies, Pareto frontier estimates (for multi-metric SmartChoices), accuracy and precision plots (for boolean metrics), and comparison to other policies (e.g., CPE). Users can configure validation criteria to automatically determine whether the new policy is fit to serve production traffic. Table 1 shows how these analyses improve production readiness.

**Behavior and performance information are displayed on an automatically generated web front-end.** The frontend shows: *(i)* training progress; *(ii)* logged distributions for each context, arm, and feedback field over time and for each policy; *(iii)* analysis results for each policy.

**Policy rollouts are managed via human-readable "tags".** Each tag references a single policy; the referenced policy can change 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*. SmartChoices users typically use the "live" policy, but custom tags enable them to use SmartChoices within the framework of existing integration testing, canary, release and rollback processes. For example, users can specify environment-dependent policy tags, allowing policy updates to be first deployed in a staging environment before being used for all traffic.

## 5 Evaluation & Case Studies

**Production Readiness.** Our service implementation mitigates many of the novel types of technical debt and production risks ML can create (see §1) by design. We quantify this in Table 1.

**Case Studies.** A wide range of problems fit into the contextual bandits framework (§2). The low latency of SmartChoices deployments enables ML to be applied towards low-level problems, and its guaranteed safety and reliability gives engineers the confidence to deploy ML in mission-critical settings. As evidence, we share a representative sample of four systems deployments in Appendix A.

## Appendix A Case Studies

***Learned Cache Eviction.*** SmartChoices was used to reduce the fraction of user-requested bytes missed in a large-scale video Content Delivery Network (CDN) cache by 9.1% at peak traffic by improving its eviction policy [26]. The low-latency inference and continuous training provided by SmartChoices was crucial in enabling this deployment.

***Optimizing Compilation.*** SmartChoices achieved performance gains in the XLA compiler by optimizing tile size selection [22] for high-level operations (HLO). Instead of evaluating all possible tile sizes for each HLO, we evaluate only the most promising 1% of candidates using SmartChoices. This lets us achieve $90 - 95\%$ of the speedup achieved by a full exhaustive search and is about 20 times faster overall. More detail on context features and a discussion of the differences between this optimization approach and existing search-based techniques (e.g., TVM) are available at [21].

***Optimizing Thread Counts.*** SmartChoices reduced tail latency for end-user queries on a flight booking search service by optimizing thread count for parallelizable stages (e.g., processing all relevant sequences of flights). SmartChoices dynamically selected the thread count per query based on context features (e.g., number of flight sub-sequences and the source and destination regions). For each query, latency and CPU usage is provided as feedback, with SmartChoices minimizing a weighted linear combination (§C) of the two. At launch, average latency reduced by 25% and 99th percentile latency reduced by 16%, without a significant increase in CPU cost.

***Optimizing Work Partitioning.*** SmartChoices improved data freshness for a monitoring service via dynamic work rebalancing between tasks [14]. Each task retrieves & summarizes telemetry for a list of workloads. Overly large tasks are terminated when they reach an execution deadline $t_d$, resulting in stale data for any unprocessed workloads. Conversely, overly small tasks incur unnecessary overhead. For each task, SmartChoices chooses whether to `shard` the list of workloads (with 0 reward) or `execute` (with reward $t_d - t_e$, where $t_e$ is actual execution time); the reward formulation encourages sharding only large tasks. Context features include workload count, day of week, and metadata on retrieved data sources. Using SmartChoices reduced the fraction of tasks hitting $t_d$ by 53% and significantly reduced alerts on stale data, translating directly into time savings for the service's owners.

## Appendix B Related Work

Contextual bandits have been applied across industry to a wide range of problems, including personalizing products [4], improving recommendations [10, 17, 24], and responding well in ambiguous contexts [19, 23]. SmartChoices has been successfully applied to similar problems; however, its low latency also enables applications to the systems problems discussed in §A.

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, though it would be straightforward to add.

Carbune et al. [9] 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 RL, assorted safety features, and a service architecture. Associated semantics for SmartChoices were analyzed in Abadi & Plotkin [1].

Several other projects have explored how to effectively deploy ML for decision making within production systems. We summarize key differences in Table 2, with additional details below.

**Microsoft's Decision Service** [2] is most similar to SmartChoices, sharing many design principles & goals. SmartChoices' lower decision latency makes it better suited for low-level system optimizations. **ReAgent** [11] (prev. Horizon) is a platform for deploying RL in production, with similar goals to SmartChoices. SmartChoices provides greater ease of use (e.g., providing more automation for data preprocessing, training, and model updates). **Looper** [18] is a platform for optimizing over "product goals": metrics that summarize the aggregate effect of many decisions. Looper uses both per-decision metrics and product goals to optimize over a class of parameterized "strategy blueprints" using a sequence of A/B tests selected via BO. Unlike Looper, SmartChoices does not have a direct notion of product goals and is not tightly integrated with an A/B experiment framework. However,

|  | DS | R | L | PSS | SC |  |
|---|---|---|---|---|---|---|
| *Multimetric Optimization* |  |  | ✓ |  | ✓ | (§C) |
| *Built-in Fallback Logic* |  |  |  |  | ✓ | (§3.2) |
| *Local Inference* | ✓ |  |  | ✓ | ✓ | (§4) |
| *Training on Data from Multiple Machines* | ✓ | ✓ | ✓ |  | ✓ | (§4) |
| *Policy Rollouts can be Gated on CPE* | ✓ | ✓ | ✓ |  | ✓ | (§4) |
| *Median Inference Latency (reported)* / µs | $200^\dagger$ | N/A | $2000^\ddagger$ | 0.004 | 10 | (§4) |

Table 2: A comparison of the key features of SmartChoices & other projects deploying ML for decision making within production systems. Lower decision latency makes projects better suited for low-level system optimizations. $^\dagger$: average latency; $^\ddagger$: excludes feature extraction latency of 45 ms. *R: ReAgent, L: Looper, DS: Decision Service, PSS: Prediction System Service, SC: SmartChoices.*

optimizing per-decision metrics that are a proxy for the product goal (combined with a grid search over scalarization parameters in the multimetric case) has typically sufficed. Last, the **Prediction System Service** [29] focuses on low-latency predictions for systems applications. It is similar to our in-process implementation (§4), with logging and training occurring on a single machine. However, in contrast to SmartChoices, it focuses on predictions, without an explicit notion of data gathering or exploration. Using service SmartChoices also enables training on data from an entire cluster.

Finally, Natarajan et al. [20] present *programming by rewards* (PBR), which synthesizes decision functions as if-then-else programs that are checked in. This has advantages in terms of interpretability, speed, and avoiding additional dependencies; however, there are no affordances for adapting policies to changing environments over time. In addition, the class of policies supported are restricted.

## Appendix C Multimetric optimization

**Metric Constraints**. SmartChoices can optimize with respect to one metric while constraining our decisions with respect to the others, e.g., maximize reward $r : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ while keeping the average cost $c : \mathcal{X} \times \mathcal{A} \to \mathbb{R}^{k-1}$ below a target $C \in \mathbb{R}^{k-1}$, over a random stream of (context, arm set) pairs:

$$\pi^* := \arg\max_\pi \left\{ \mathbb{E}_{(X,A)} \left[ r(X, \pi(X, A)) \right] \ : \ \mathbb{E}_{(X,A)} \left[ c(X, \pi(X, A)) \right] \leq C \right\}$$

Without contexts (i.e., $\mathcal{X} = \emptyset$), this is Bayesian optimization with unknown constraints [12]. With contexts, it is closely related to contextual bandits with knapsack constraints, which has known results for stochastic [3] and adversarial settings [27]. In contrast to prior work, we are interested in *average budgets* (e.g., serving an infinite stream of requests at bounded average latency) rather than cumulative budgets that run out (e.g., dynamically pricing a limited supply of goods for maximum revenue).

Note costs, like rewards, must be *learned*; hence constraint violations during learning cannot be fully avoided. While there are ways to mitigate this, SmartChoices is not designed or intended for circumstances where individual decisions are high-stakes (e.g., selecting medical treatments).

**Scalarization**. For applications with soft constraints, SmartChoices supports exploring the *Pareto frontier* of achievable metric combinations by *scalarizing* predicted metrics, *i.e.,* combining them in a single scalar reward via a known *scalarization* function. SmartChoices users can select from parameterized scalarizations at inference time. Rapidly changing between diverse scalarizations induces efficient exploration of the Pareto frontier. Linear scalarizations (*i.e.,* linear combinations of metrics) are simple and work well when the achievable metrics form a convex set. For generally shaped Pareto frontiers, we use the hypervolume scalarizations [28], which can discover arbitrarily shaped frontiers. After investigating the metric frontier, SmartChoices users may fix a preferred tradeoff (*i.e.,* scalarization parameters) and easily deploy the corresponding policy.

Decoupling scalarization from metric predictions allows us to largely reduce multiobjective optimization to the single objective case, which simplifies both the infrastructure and the algorithmics.

## Appendix D In-Process Implementation

Users typically find a service deployment sufficient, and choose in-process deployments in extremely low-level settings with low latency requirements ($<$1 µs), where network usage is limited, or where fast policy updates ($\sim$200 ms) are critical.

## References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Bazel. Rules | bazel, Feb 2023. URL `https://bazel.build/extending/rules`.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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-google-cloud-dataflow`.

[15] Kubeflow. Kubeflow, Sep 2023. URL `https://github.com/kubeflow/kubeflow`.

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

[17] 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`.

[18] 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`.

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] Song, Z., Chen, K., Sarda, N., Altınbüken, D., Brevdo, E., Coleman, J., Ju, X., Jurczyk, P., Schooler, R., and Gummadi, R. {HALP}: Heuristic aided learned preference eviction policy for {YouTube} content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 1149–1163, 2023.

[27] 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.

[28] 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.

[29] Zhang, Z., Glova, A. O., Sherwood, T., and Balkind, J. A prediction system service. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 48–60, 2023.