AdaChain: A Learned Adaptive Blockchain

Chenyuan Wu
University of Pennsylvania
wucy@seas.upenn.edu

Bhavana Mehta
University of Pennsylvania
bhavanam@seas.upenn.edu

Mohammad Javad Amiri
University of Pennsylvania
mjamiri@seas.upenn.edu

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Abstract

This paper presents *AdaChain*, a learning-based blockchain framework that adaptively chooses the best permissioned blockchain architecture to optimize effective throughput for dynamic transaction workloads. AdaChain addresses the challenge in Blockchain-as-a-Service (BaaS) environments, where a large variety of possible smart contracts are deployed with different workload characteristics. AdaChain supports automatically adapting to an underlying, dynamically changing workload through the use of reinforcement learning. When a promising architecture is identified, AdaChain switches from the current architecture to the promising one at runtime in a secure and correct manner. Experimentally, we show that AdaChain can converge quickly to optimal architectures under changing workloads and significantly outperform fixed architectures in terms of the number of successfully committed transactions, all while incurring low additional overhead.

1 Introduction

Permissioned blockchain systems have enabled a new class of data center applications, ranging from contact tracing [56], crowdworking [14], supply chain assurance [15, 60], and federated learning [57]. The popularity of these services has motivated cloud providers, e.g., Amazon [2, 3], IBM [8], Oracle [9], and Alibaba [65], to offer *Blockchains-as-a-Service* (BaaS) [26].

BaaS offerings have resulted in a large variety of possible smart contract deployments. Different smart contracts may exhibit different workload characteristics, such as read/write ratios, skewness of popular keys, compute intensity, etc. To address these variations in workloads, there has been a proliferation of permissioned blockchain systems, e.g., Tendermint [46], Fabric [16], Fabric++ [59], Fabric# [58], Streamchain [39], and ParBlockchain [12]. These systems present significant variation in architectural design, including the number of transactions in a block, stream processing (with no blocks), the use of reordering and early aborts, and the sequence in which ordering, execution and validation are done.

Past studies [22, 33] have shown that different blockchain architectures and hyperparameter settings are optimal for different workloads with varying properties (e.g. system load, write ratios, skewness, and compute intensity). We experimentally confirmed this observation. Figure 1 shows the performance of various architectures across four different workloads, showcasing significant variations in throughput. For example, for Workload A[1], which is highly compute-intensive, an Execute-Order-Validate (XOV) architecture with reordering provides the best throughput. On the other hand, for Workload D, which requires significantly less computation but has higher skewness, an Order-Parallel Execute (OXII) architecture demonstrates the highest throughput. This clearly shows the dependency between workload characteristics and the optimal blockchain architecture for each workload.

Currently, BaaS providers must choose a single architecture to offer customers, potentially resulting in poor performance, as no single architecture provides dominant throughput. Even when the user has control over the blockchain architecture, choosing the right architecture and parameters is not easy given the large configuration space. Moreover, in a BaaS setting, the workload may fluctuate and change, as different tenants scale up or down their smart contracts deployments, and client requests fluctuate with different patterns throughout the day. Of course, one could imagine building a static mapping from workload characteristics to optimal blockchain architectures – but this mapping would (1) be expensive to compute, (2) depend on the underlying hardware, (3) still be suboptimal for workloads that shift unexpectedly over time, and (4) require recomputing the mapping each time a new blockchain architecture is developed.

In this paper, we propose *AdaChain*, a reinforcement learning-based blockchain framework that chooses the best blockchain architecture and sets appropriate parameters in order to maximize effective throughput for dynamic transaction workloads. Experimentally, we show that AdaChain is not only able to select optimal or near-optimal configurations for a wide variety of workloads, but its reinforcement learning approach also allows it to quickly adapt to new hardware, new storage subsystems, and new unanticipated workload changes on the fly.

In order to build an adaptive blockchain, AdaChain relies on two key innovations. First, it models the selection of a blockchain architecture as a contextual multi-armed bandit problem, a well-studied reinforcement learning problem with asymptotically optimal results [42]. This formulation allows AdaChain to apply classical algorithms, such as Thompson sampling [23], to select blockchain architectures in a way that minimizes *regret* (the difference between the performance of the chosen architecture and the optimal architecture). AdaChain will strategically test different architectures to learn which ones are well-suited to the user's workload. It learns which architectures work best by observing the characteristics of the workload and the effective throughput of the system. When the workload changes, AdaChain notices drops in throughput, and can automatically adjust the blockchain architecture and parameters to maximize performance, all without any user intervention.

Second, AdaChain introduces protocols to switch from one blockchain architecture to another in a live system, while maintaining

[1]Details about each workload can be found in Tables 2 and 3.

strong serializability properties. This switching protocol is not only required for AdaChain to function (multi-armed bandits generally require making multiple decisions before the optimal is reached), but also enables a new class of blockchains that can more-or-less seamlessly transition between different architectures to support the shifting workloads in the real-world. Intuitively, the switching protocol works by splitting switching decisions between two paths. In the normal path, all nodes agree to switch to the same new architecture after a certain number of blocks have been committed, while in the slow path, all nodes switch to the same architecture after failing to make progress on processing transactions for a certain amount of time.

Specifically, this paper makes the following contributions.

- **Learned adaptive blockchain.** To the best of our knowledge, AdaChain is the first blockchain system to support *automatically adapting to an underlying, dynamic workload*. Through careful modeling of the states, actions, and objective function, AdaChain's use of reinforcement learning makes it the first blockchain system to *learn from its mistakes* and self-correct.
- **Multi-architecture switching.** Additionally, we also present the first blockchain system capable of switching from one architecture to another at *runtime* while respecting correctness and security concerns.
- **Analysis of architecture impact on blockchain performance.** We perform a large-scale measurement examining the relationship between architecture choice and blockchain performance. We implemented a wide range of blockchain architectures, and through a suite of workload parameters, we identified architecture configurations and runtime settings that significantly impact performance improvements. Our experiments highlight the large state space, which renders manual heuristics difficult to achieve. The workloads, architectures and measured performance will be publicly available to aid future research.
- **Prototype and performance evaluation.** We have developed a prototype of AdaChain, which will be publicly available under an open-source license. Our evaluation results on CloubLab demonstrate that AdaChain can converge quickly to optimal architectures under changing workloads, significantly outperform existing fixed architectures, and incur low additional overhead.

2 Architecture Landscape

To motivate AdaChain, we first examine, both intuitively and experimentally, why different blockchain architectures perform diversely across different workloads. In Section 2.1, we highlight a number of blockchain architectures, and illustrate their advantages and disadvantages. The point here is not that some blockchain architectures are "better" or "worse" than others, but rather that each blockchain architecture performs well under some conditions and poorly under others. In Section 2.2, we argue that a blockchain that can adaptively switch between multiple architectures is able to achieve "best of all worlds" performance.

2.1 Blockchain Architectures and Workloads

Table 1 lists representative blockchain systems and their architectures, where the design space consists of seven performance optimizations (P1-P7) and two correctness dimensions (C1-C2). Figure 1 shows their corresponding performance under four different

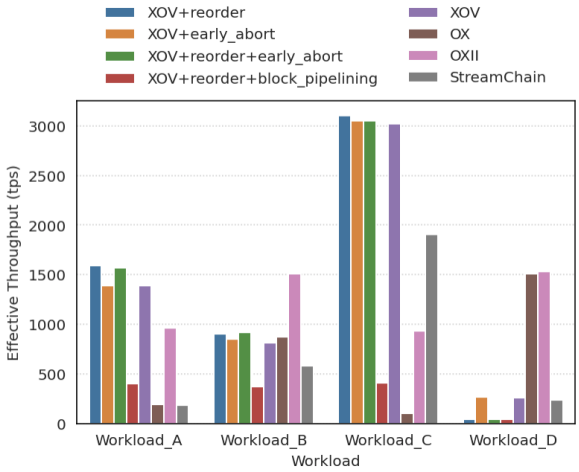


Figure 1: Performance of different blockchain architectures under various workloads. The performance of different architectures can vary significantly between workloads. Workloads and architectures are described in Table 1 and 2.

workloads. Here, we use *effective* throughput as the performance metric, which measures the number of successfully committed transactions per second.

The workloads `A` to `D` are characterized in Table 2. BaaS workloads embody a large extent of variations. For instance, different transactions might invoke different percentages of write operations to the underlying key-value store, as represented by the *write ratio*. These transactions might also contend to access or update the same set of popular keys (or hot keys), as indicated by the *contention level*. In addition, the runtime *load* on a BaaS can be determined by the frequency of issued transactions by each client and the number of active clients varying with time. Last, *compute intensity* is an important characterization of BaaS workloads, as pointed out by [36, 58, 64]. This is because permissioned blockchains support a wide range of applications, some of which are compute-intensive (e.g., those that provide security and correctness guarantees for machine learning applications).

Below, we briefly describe the design principles of each architecture and explain the intuition behind why the performance of each architecture can vary under different workloads.

Order-Execute (OX). The order-execute architecture has been widely used in permissioned blockchain systems such as Tendermint [46], Quorum [24], Chain Core [4], Multichain [37], Hyperledger Iroha [7], and Corda [5]. In the OX architecture, transactions are totally ordered and batched into blocks and then transactions of a block are executed sequentially. As a result, the OX architecture does not require a Multi-version Concurrency Control (MVCC) validation phase, which is used to resolve conflicts between transactions, and hence, no transactions will be aborted due to conflicts. As shown in Figure 1, this design principle makes OX outstanding at workload `D`, where transactions are write-heavy and contentious, i.e., transactions update a small set of hot keys. On the other hand, OX performs comparatively poorly on workloads `A` and `C`, which are compute-intensive. Due to the lack of parallel execution mechanisms, OX cannot take advantage of the multi-core processing power of modern servers.

Architecture	Rep. System	P1. Block Size	P2. Early Exec.	P3. Dependency Graph	P4. Early Abort	P5. Cross-Block Conflicts	P6. Parallel Exec.	C1. MVCC	C2. Isolation
OX	Tendermint [46]	tunable	✗	✗	✗	-	✗	✗	strong serializable
OXII	ParBlockchain [12]	tunable	✗	✓	✗	-	partial	✗	strong serializable
XOV	Fabric [16]	tunable	✓	✗	✗	V	fully	✓	strong serializable
XOV++	Fabric++ [59]	tunable	✓	✓	X, O	V	fully	✓	strong serializable
XOV#	Fabric# [58]	tunable	✓	✓	O	O	fully	✗	serializable
XOV	StreamChain [39]	1	✓	✗	✗	V	fully	✓	strong serializable

Table 1: Comparing design principles of existing permissioned blockchain architectures. Here, P stands for performance, C stands for correctness, X stands for execution, O stands for ordering, and V stands for validation.

Workload	Write Ratio	Contention Level	Load	Compute Intensity
A	low	high	high	high
B	moderate	high	moderate	low
C	moderate	low	high	very high
D	high	very high	moderate	very low

Table 2: Characterizing workloads A, B, C and D. Specific workload parameters are presented in Table 3.

Order-Parallel Execute (OXII). In the OXII architecture, used by ParBlockchain [12], transactions are first totally ordered and batched into blocks. OXII then constructs a dependency graph for transactions within a block based on their positions. Specifically, if t_i is ordered before t_j, and the pair of transactions are conflicting, OXII adds an edge from t_i to t_j. This dependency graph is then used to execute transactions in parallel, i.e., a transaction can be executed once all its predecessors have finished execution. Given a higher level of execution parallelism than OX, OXII performs better than OX under computation-heavy workloads such as A and C. Note that even for a given workload, OXII requires careful tuning of block size; a large block results in high overhead in dependency graph construction, while a small block results in less parallelism and higher communication overhead.

Execute-Order-Validate (XOV). Hyperledger Fabric [16] presents the XOV architecture (which was first introduced by Eve [41] in the context of Byzantine fault-tolerant SMR) by switching the order of the ordering and execution phases such that transactions are simulated fully in parallel before being ordered in the ordering phase. Since it utilizes early execution, XOV requires an MVCC validation phase to invalidate all transactions that are simulated on stale data, and commits only the validated transactions to the world-state and the blockchain ledger. This early execution enables XOV to perform well on contention-free workloads such as C. On the other hand, XOV demonstrates poor performance under contentious and write-heavy workloads, such as B and D, due to the high percentage of invalidated transactions. Similarly, as network delay increases, XOV suffers from inconsistent world states among peers as well as stale reads, and thus significantly degraded performance [22].

XOV with early abort and reordering (XOV++). The XOV++ architecture, as introduced in Fabric++ [59], follows the XOV paradigm but with some modifications. First, a dependency graph is constructed in the ordering phase to capture RW conflicts between each pair of transactions within the same block. When the graph is constructed, all elementary cycles in the graph are aborted greedily. Unlike OXII, which utilizes the graph for concurrency control, XOV++ uses the graph for transaction reordering; when there is a RW conflict between t_i and t_j, it (re)orders t_i before t_j in the block. Second, it adopts early abort techniques in both the simulation and ordering phases. Whenever XOV++ detects that a transaction operates on stale data, XOV++ immediately aborts that transaction without waiting for the final MVCC validation. As an effect of transaction reordering, XOV++ has outstanding performance on workload A, where the conflicts are reconcilable given a low write ratio. On the other hand, XOV++ performs poorly on workload D with a near-zero effective throughput. This is because, under a contentious and update-heavy workload, very few conflicts can be reconciled through reordering. Moreover, reordering becomes more expensive when there are a large number of cycles in the dependency graph, resulting in more pending blocks and, thus, more transactions that simulate on stale data.

XOV with serializable isolation (XOV#). The XOV# architecture, presented in Fabric# [58], is mainly different from XOV and XOV++ in that XOV# is serializable, while XOV and XOV++ are strong serializable. To achieve this isolation level, XOV# incrementally constructs a dependency graph that keeps track of all dependencies, including those that span across blocks in the ordering phase. Once a transaction is ordered, XOV# immediately drops this transaction if there is a dependency cycle involved. The resulting acyclic schedule is guaranteed to be serializable, thus, no extra MVCC validation is needed in XOV#. To ensure a fair comparison with other architectures, we run XOV# under the strong serializability isolation level while keeping the remaining design dimensions the same as the original XOV# (the XOV+reorder+block_pipelining bar). XOV# performs worse than vanilla XOV in all workloads A to D due to the overhead of maintaining a large dependency graph and detecting cycles. This suggests that the performance improvement reported in Fabric# is mainly due to a more relaxed isolation level.

Stream XOV. StreamChain [39] switches from block processing to stream transaction processing. Specifically, StreamChain follows the XOV paradigm while fixing the block size to 1. The motivation behind stream processing is simple: while the original, permissionless blockchains were forced to used proof of work (PoW) consensus techniques to maintain fault tolerance, a permissioned blockchain environment allows more efficient consensus protocols to be used. Thus, stream processing can reduce transaction latency. In terms of effective throughput, StreamChain has relatively good performance when the workload is lightweight or not contentious, such as in workloads B and C. Otherwise, the high block construction overhead in terms of cryptographic operations and excessive disk I/Os leads to a large number of pending blocks in StreamChain, making incoming transactions simulate on stale data. As expected, StreamChain is more sensitive to the type of storage hardware used

than OX: the system have poor performance without RAM disk [22].

StreamChain also highlights that the *parameters* of a given architecture can impact performance. A large block size leads to higher block formation overhead and latency, while a small block size results in higher communication and disk overhead.

2.2 The Case for Reinforcement Learning

The main takeaways of the previous subsection are as follows: depending on workload and hardware characteristics, the performance of a given blockchain architecture can vary drastically. We thus argue that *there is no one-size-fits-all architecture*. Figure 1 not only shows that there is no single dominant architecture, but also some architectures that perform well on one workload can end up performing quite poorly on another.

One may consider the design of simple heuristics to map workload characteristics and hardware features to the optimal blockchain architecture. However, designing a good heuristic is difficult and cumbersome in the case of permissioned blockchains. For example, Hyperledger Fabric [16] has proposed that "when the contention level is high, use OX; otherwise, use XOV". Unfortunately, such a simple heuristic leads to wrong decisions in 50% of our recorded traces, let alone in a real BaaS production environment.

This suggests, to develop a good heuristic, an expert needs to exhaustively experiment over the entire state and action space to understand the intricate interactions. The expert also needs to carefully tune the specific threshold parameters that distinguish "very high contention" from "high contention", or "low write ratio" from "moderate write ratio", etc. Given the large space of workload (i.e., write ratio, contention level, load, compute intensity) and hardware (i.e., CPU, RAM, disk, network) features, if the expert discretizes each dimension by uniformly sampling 5 points in it and tests each architecture for 5 seconds, it takes at least 90 days to conduct the experiments even when the effect of block size is ignored. This is a conservative estimate given that most dimensions will require more than 5 sample points in practice. Due to the lack of blockchain simulators that capture the variations in architectures, such experiments are hard to be conducted in parallel.

Moreover, such heuristics become outdated when new hardware or blockchain architectures are introduced in the BaaS and hence, difficult for the expert to keep up with these changes. For instance, when the BaaS provider introduces non-volatile RAM [61] to replace the traditional combination of volatile RAM and disk, or when BaaS transitions to disaggregated architecture [64] where there is extra latency for accessing memory, new hardware features are also naturally introduced. Thus, the expert needs to rerun all experiments over these new hardware to understand how they interact with workloads and blockchain architectures.

Reinforcement learning (RL) is an ideal solution to this Sisyphean task, which has shown superior performance in other learned systems [51, 52]. Unlike supervised learning that assumes training data is complete and requires a separate data collection process prior to deployment, RL-based system learns from its mistakes and optimizes long term rewards through its trials. With reinforcement learning, AdaChain can optimize itself to whatever workloads, hardware and blockchain architectures at hand, providing significant *operational benefits*.

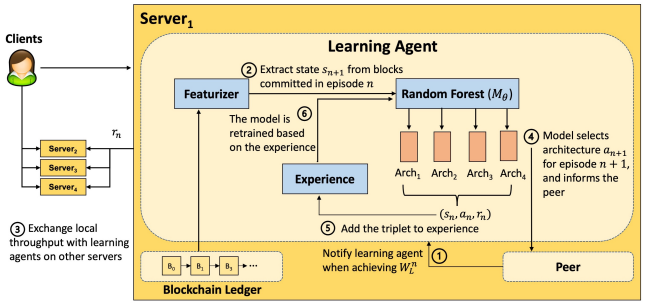


Figure 2: Architecture of AdaChain. For readability, we only present the internals of server$_1$.

3 AdaChain Overview

At a high level, AdaChain contains two key components: a machine learning model (the *learning agent*) which guides AdaChain towards better and better blockchain architectures, and an *architecture switching* mechanism that allows AdaChain to near-seamlessly transition from one blockchain architecture to another while ensuring correctness and security.

Learning agent. AdaChain's learning agent models the problem of selecting a blockchain architecture as a contextual multi-armed bandit (CMAB) problem [66]: periodically, AdaChain examines the most recent properties of the workload (*context*), and then selects one of many blockchain architectures (*arms*). After making the selection, it observes the effective throughput of the newly selected architecture (*reward*). To be successful, AdaChain must balance the *exploration* of new, untested architectures with *exploiting* past experience to maximize throughput – without a careful balance of exploration and exploitation, AdaChain risks failing to discover an optimal configuration (too much exploitation), or performing no better than random (too much exploration). We select this CMAB formulation (as opposed to generalized reinforcement learning models) because CMABs are exceptionally well-studied, and many asymptotically-optimal algorithms exist to solve them [10, 23].

Details about the learning agent are provided in Section 4.

Switching architecture. AdaChain utilizes a switching protocol that allows it to switch from one blockchain architecture to another in a distributed fashion across all nodes in the blockchain deployment, while transactions are ongoing. AdaChain achieves this by splitting switching decisions between two paths, a normal path in which all nodes agree to switch to the same new architecture after a certain number of blocks have been committed, and a slow path in which all nodes switch to the same new architecture after failing to make progress for a certain amount of time.

Details about AdaChain's switching protocol are in Section 5.

AdaChain workflow overview. Figure 2 shows the overall architecture of AdaChain. AdaChain operates in *episodes*, where within one episode, the blockchain architecture remains unchanged. When the learning agent finds an architecture candidate, it instructs the peer to use that architecture for the next episode. Each episode is marked by the completion of a constant number of transactions ($\Delta N_{\text{episode}}$), including invalidated transactions. This episode design ensures AdaChain does not stuck in a bad architecture for a long time even when the fraction of invalidated transactions is high

due to conflicts. Below, we describe how the learning agent proposes the architecture for episode $n+1$ in detailed steps. Although our discussion below focuses on the internals of server_1, the same procedure happens simultaneously on every blockchain server.

Step 1: Notifying the learning agent. In episode n, the peer notifies its local learning agent when the number of committed blocks has reached a certain watermark. The notification also includes the local performance measurement r_n in episode n.

Step 2: Featurization. Since learning agents are distributed across different servers, the state (i.e., some features that capture the workload) that they need in order to make a decision should also be distributed. In AdaChain, states are not only distributed, but also decentralized as no single entity controls the state. This is possible with negligible overhead due to two key insights: (1) the blockchain ledger contains rich information about the workload, and thus is a good source of raw data for featurization; (2) the ledger is naturally decentralized and consistent across peers. Thus, once the agent is notified by the peer, its featurizer extracts the state s_{n+1} from blocks committed in episode n. Features are described in detail in Section 4.2.

Step 3: Exchanging performance measurements. The locally observed performance is different across different peers, and malicious peers could even manipulate the local measurement. To ensure each honest server has the same architecture for episode $n+1$, the learning agent on server_1 exchanges the local measurement r_n with learning agents on every other server, so as to agree on performance measurement. Details are described in Section 5 and 6.

Step 4: Estimating the performance for each architecture. The predictive model M_θ predicts the performance of each architecture candidate under state s_{n+1}, and selects architecture a_{n+1} that is predicted to have the best performance. The learning agent then informs the peer to switch to a_{n+1} for episode $n+1$.

Step 5: Building experience buffer. Once an r_n is obtained, the learning agent adds the (s_n, a_n, r_n) triplet to the experience buffer. Note that s_n and a_n are derived prior to the start of episode n.

Step 6: Retraining. The predictive model M_θ is periodically retrained based on the experience buffer, creating a feedback loop. As a result, AdaChain's predictive model improves, and AdaChain more reliably picks the best architecture for the observed state.

Assumptions. In short, AdaChain can adapt itself according to the workload and hardware setup to continually improve performance. Moreover, AdaChain is an *online* learned system that does not require a separate and cumbersome data collection process prior to deployment. Our current design makes two assumptions. First, in AdaChain, similar to many other permissioned blockchain systems [13, 24, 46], each node serves as both the ordering and execution (endorser) node. This, however, is in contrast to Hyperledger Fabric and its variants that separate endorsing and ordering roles. Second, AdaChain is designed for a homogeneous setup, where different servers have access to similar resources. While having these two assumptions in place simplifies the system design, they have been used in real-world BaaS deployments. Removing these assumptions is an avenue for future work.

4 Learning Algorithms

In this section, we discuss AdaChain's learning approach in detail. We first formalize AdaChain's learning problem as a contextual multi-armed bandit problem, and then discuss our selected algorithm, Thompson sampling, for solving such problems. We next describe the predictive model used by AdaChain, followed by the specific state and action space design.

Contextual multi-armed bandits (CMABs). In a contextual multi-armed bandit problem, an agent periodically makes decisions in a number of episodes, enumerated by n. In each episode, the agent selects an action a_n based on a provided state s_n, and then receives a reward r_n. The agent's goal is to select actions in a way that minimizes *regret*, i.e., the difference between the reward from the chosen action and the reward from the optimal action. CMABs assume that each episode is independent[2] from each other, and that the optimal decision depends only on the state s_n. As described in Section 3 and 5, in order to be responsive to workload changes, there are no pending blocks across different episodes in AdaChain. Thus, each episode in AdaChain can also be considered to be independent (although caching effects may bring a small amount of dependence between episodes).

AdaChain's formulation. AdaChain uses *effective* throughput as the performance metric P to maximize, which is the number of successfully committed transactions per second. For each episode, it must select an architecture to use. AdaChain's goal is to select the best architecture (in terms of effective throughput) in the family of available architectures A, given the current perceived workload $w \in W$. We call this selection function $S : W \rightarrow A$. We formalize the goal as a regret minimization problem, where the regret r_n for an episode n is defined as the difference between the effective throughput of the architecture selected by AdaChain and the ideally optimal architecture as presented in equation 1.

$$r_n = \max_{a \in A} P(w, a) - P(w, S(w)) \tag{1}$$

We use effective throughput as the performance metric since it is the dominant metric used by blockchain benchmark tools [6, 29, 36] and previous literatures that proposed fixed architectures [12, 16, 58, 59]. Extending the optimization goal to a combination of metrics is left for future work.

Thompson sampling. While there are many algorithms to solve contextual multi-armed bandit problems, we select Thompson sampling for its simplicity: at the start of each episode, we train a model based on our current experience, and then select the best action as predicted by the model. In order to train the model, Thompson sampling deviates from traditional ML techniques: instead of selecting the model parameters that are most likely given the data, we *sample model parameters proportionally to their likelihood given the training data*. More formally, we can define maximum likelihood estimation as finding the model parameters θ that maximize likelihood given experience E: $\arg\max_\theta P(\theta \mid E)$ (assuming a uniform prior). Instead of maximizing likelihood, Thompson sampling simply samples from the distribution $P(\theta \mid E)$. This means that if we have a lot of data suggesting that our model weights should be in

[2]Contextual multi-armed bandit algorithms have been shown to be effective even when these condition do not strictly hold [23].

a certain part of the parameter space, our sampled parameters are likely to be in the part of the space. Conversely, if we have only a small amount of data suggesting that our model weights should be in a certain part of the parameter space, we may or may not sample parameters in that part of the space during any given episode.

4.1 Predictive Model

AdaChain uses random forests [18] as the predictive model due to their good performance on data sets of moderate sizes and fast inference. The model takes the state (i.e., workload) concatenated with action (i.e., architecture choice) as input, and outputs the predicted performance.[3] Thus, given a state, AdaChain enumerates the action space and uses the model to predict the performance of each action. AdaChain then chooses the action with the best-predicted performance to be carried out. Once there is a tie on the best-predicted performance, AdaChain breaks the tie randomly to avoid local maxima.

Integrating random forests with Thompson sampling requires the ability to sample model parameters from $P(\theta \mid E)$. The simplest technique (which has been shown to work well in practice [54]) is to train the model as usual, but only on a bootstrap [17] of the training data. In other words, the random forest is trained using $|E|$ random samples drawn with replacement from experience E, inducing the desired sampling properties. AdaChain uses this bootstrapping technique for its simplicity.

4.2 State Space

In AdaChain, the state represents properties of the client workload. AdaChain captures the state space using the four features below. To ensure the accuracy of feature extraction, all aborted or invalidated transactions are still written to the ledger with a validity flag (similar to [38]). Below, we assume a window of blocks b_i to b_j in the ledger are read by the learning agent for featurizing the current state.

Write ratio. We observe that counting the write ratio in terms of write accesses to the key-value store is not effective for predicting performance. Thus, AdaChain measures the write ratio at the transaction level: once a transaction writes to the key-value store, it is viewed as a write transaction. The write ratio is the ratio of write transactions to the number of all transactions during b_i and b_j.

Hot key ratio. AdaChain measures the frequency that each key is accessed during b_i and b_j. It then takes the frequency corresponding to the hottest key to be the hot key ratio.

Transaction arrival rate. AdaChain timestamps each transaction upon its first arrival to the system. AdaChain first measures the number of all transactions from b_i to b_j, denoted as N, and then derives the transaction arrival rate using $\frac{N}{ts_j - ts_i}$, where ts_i represents the arrival timestamp of the first transaction in b_i and ts_j represents that of the last transaction in b_j.

Execution delay. AdaChain uses average execution delay of all transactions in the period of b_i to b_j.

4.3 Action Space

In AdaChain, the action space consists of different blockchain architectures. One naïve approach to represent the action space is to give each architecture a one-hot encoding. However, from the random forest's perspective, this approach makes two semantically close architectures totally unrelated, resulting in ineffective splits, and thus poor prediction accuracy. For example, assume XOV is represented by vector $(1, 0, ..., 0)$ in the one-hot encoding. Random forest might split on the first dimension in the vector, i.e., XOV is its left child, while everything non-XOV is its right child. Each child's performance will be predicted using the average performance of that child. Clearly, XOV++ and StreamChain might have a relatively close performance to XOV, but they will always fall into a wrong child node and their predicted performances are wrongly averaged.

Thus, AdaChain chooses to first featurize the blockchain architectures to maintain the semantic information of their design. Feature engineering an optimal representation of blockchain architectures is a difficult and inexact task. Instead of attempting to design an all-encompassing representation that captures every dimension of blockchain architectures, we instead selected a simple representation based on our intuition of the most important properties. We leave investigating alternative representations to future work.

AdaChain therefore captures the action space using three main features: block size, early (speculative) execution, and dependency graph construction. Block size is a scalar variable, representing the number of transactions within a block. The block size in AdaChain is also equal to the batch size in the consensus protocol. To limit the growth of action space, the block size can not exceed $1,000$, which is larger than typical block sizes used in blockchain systems, and we further discretize the block size using paces. Early execution and dependency graph construction are both binary variables. Thus, AdaChain's action space consists of 100 choices in total.

We do not consider parallel execution as a feature because it can be derived from the two previous features (i.e., early execution and dependency graph construction): (1) early execution of transactions happens fully in parallel; (2) the goal of constructing a dependency graph is to execute independent transactions in parallel.

5 Switching Architectures

This section discusses the architecture switching mechanism of AdaChain. We first introduce the normal path of operations, followed by our timeout-based mechanism in the slow path.

5.1 Normal Path

Algorithm 1 presents the normal path of operations. Each server in AdaChain runs Algorithm 1 in a distributed fashion in order to carry out architecture switching. Here, S is the set of blockchain servers, i stands for the index of the server, n stands for the current episode, and $\Delta N_{\text{episode}}$ and ΔN_{learn} are two constant hyperparameters. At a high level, the normal path introduces two watermarks: a low watermark (W_L^n) that triggers the learning phase, and a high watermark (W_H^n) that marks the end of an episode.

The untrustworthiness of participants in a blockchain system prevents us from relying on a centralized entity to featurize the state and measure the reward. Thus, inspired by the PBFT protocol [20], AdaChain conducts them in a decentralized fashion. Upon reaching the low watermark W_L^n, each server $i \in S$ records its locally observed throughput p_i^n of episode n and featurizes the state for the next episode $n+1$ from its local blockchain ledger (lines 1-3). Although most dimensions of the state are naturally consistent across different servers, there can be slight variations on the

[3] This corresponds to a *value based model*. A *policy* model, in which the predictive model predicts simultaneously the probability of each action being optimal, might be an interesting direction for future work.

Algorithm 1 Normal path

▷ On each server i
1: **Upon** index of local last committed block b_{last} reaching W_L^n
2: Record performance p_i^n
3: Extract features $f_i^{n+1} = (w_i^{n+1}, c_i^{n+1}, r_i^{n+1}, e_i^{n+1})$ from block W_H^{n-1} to W_L^n
4: Multicast $\langle$CHECKPOINT, $n, i, e_i^{n+1}, p_i^n\rangle_{\sigma_i}$ to all servers
▷ On the leader sever l
5: **Upon receiving** valid CHECKPOINT messages from a quorum Q of $2f+1$ servers
6: Compute $e^{n+1} \leftarrow \text{median}\{e_j^{n+1} | j \in \text{Q}\}$
7: Compute $p^n \leftarrow \text{median}\{p_j^n | j \in \text{Q}\}$
8: Multicast $\langle\langle$PROPOSE, $e^{n+1}, p^n\rangle_{\sigma_l}, C\rangle$ to all servers
▷ On each server i
9: **Upon receiving** a PROPOSE message from the leader
10: **if** e^{n+1} and p^n are valid (based on C) **then**
11: Multicast $\langle$ACCEPT, $n, i, e^{n+1}, p^n\rangle_{\sigma_i}$ to all servers
12: **Upon receiving** valid matching ACCEPT messages from $2f+1$ different servers
13: Multicast $\langle$COMMIT, $n, i, e^{n+1}, p^n\rangle_{\sigma_i}$ to all servers
14: **Upon receiving** valid matching COMMIT messages from $2f+1$ different servers
15: Add p^n to experience and derive action a_{n+1} based on f^{n+1}
16: **if** T^n transactions have been committed **then**
17: Abort any new incoming transaction t in the ordering phase
18: **Upon** b_{last} reaching W_H^n
19: Pause block formation thread until action a_{n+1} is derived
20: $W_L^{n+1} \leftarrow W_H^n + \lfloor \Delta N_{\text{learn}} / |b_{n+1}| \rfloor$
21: $W_H^{n+1} \leftarrow W_H^n + \lfloor \Delta N_{\text{episode}} / |b_{n+1}| \rfloor$
22: $T^{n+1} \leftarrow T^n + \lfloor \Delta N_{\text{episode}} / |b_{n+1}| \rfloor \times |b_{n+1}|$
23: $n \leftarrow n+1$
24: Carry out action a_{n+1}
25: Reset timer τ

Algorithm 2 Slow path

▷ On each server i
1: **if** timer τ timeouts and b_{last} has not reached W_L^n **then**
2: pause block formation thread after committing the current block
3: Record performance p_i^n
4: Multicast $\langle$S-CHECKPOINT, $n, i, b_{\text{last},i}^n\rangle_{\sigma_i}$ to all servers
▷ On each server j where τ has not been expired
5: **Upon receiving** $f+1$ valid S-CHECKPOINT messages from different servers
6: pause block formation thread after committing the current block
7: Record performance p_j^n
8: Multicast $\langle$S-CHECKPOINT, $n, j, b_{\text{last},j}^n\rangle_{\sigma_j}$ to all servers
▷ On the leader sever l
9: **Upon receiving** valid S-CHECKPOINT messages from a quorum Q of $2f+1$ servers
10: Compute $W_H^n \leftarrow \max\{b_{\text{last},j}^n | j \in \text{Q}\}$
11: Multicast $\langle\langle$S-PROPOSE, $W_H^n\rangle_{\sigma_l}, C'\rangle$ to all servers
▷ On each server i
12: **Upon receiving** a S-PROPOSE message from the leader
13: **if** W_H^n is valid (based on C') **then**
14: Multicast $\langle$S-ACCEPT, $n, i, W_H^n\rangle_{\sigma_i}$ to all servers
15: **Upon receiving** valid matching S-ACCEPT from $2f+1$ different servers
16: Multicast $\langle$S-COMMIT, $n, i, W_H^n\rangle_{\sigma_i}$ to all servers
17: **Upon receiving** valid matching S-COMMIT from $2f+1$ different servers
18: Resume block formation thread
19: Extract features $f_i^{n+1} = (w_i^{n+1}, c_i^{n+1}, r_i^{n+1}, e_i^{n+1})$ from block W_H^{n-1} to b_{last}
20: Multicast $\langle$CHECKPOINT, $n, i, f_i^{n+1}, p_i^n\rangle_{\sigma_i}$ to all servers
▷ On the leader sever l
21: **for** every transaction t in the ordering phase **do**
22: $t.episode \leftarrow n$
▷ On each server i
23: **for** every transaction t committed by consensus **do**
24: **if** $t.episode \neq n$ **then**
25: abort t

execution delay, e_i^{n+1}, and measured throughput. Thus, each server i multicasts a checkpoint message consisting of e_i^{n+1} and p_i^n to all other servers (line 4). AdaChain relies on the leader server to (1) collect a quorum Q of $2f+1$ checkpoint messages, (2) calculate the median of observed throughput values to be the global reward p^n, and (3) calculate the median of the execution delay values e^{n+1} to be part of the global state (lines 5-7). Once both values are computed, the leader multicasts a propose message, including the values and the set C of $2f+1$ received checkpoint messages to all servers (line 8). Upon receiving the propose message, each server validates the message according to the set C and multicasts an accept message to all other servers (lines 9-11). Each server then waits for $2f+1$ matching accept messages before sending a commit message (lines 12-13). The accept and commit phases, similar to prepare and commit phases of PBFT, ensure that values are correct and replicated on a sufficient number of nodes. Finally, when a server receives $2f+1$ commit messages, the predictive model will derive action a_{n+1} as described in Section 4 (lines 14-15). Note that since accept and commit messages are broadcast to all servers, even if a server has not received the propose message from the leader (due to the asynchronous nature of the network or the maliciousness of the leader), the server still has access to the values.

In order to be responsive to workload changes, each episode is marked by the completion of $\lfloor \Delta N_{\text{episode}} / |b_n| \rfloor$ blocks, where $\Delta N_{\text{episode}}$ is a constant hyperparameter of the system (10,000 transactions in the current deployment) and $|b_n|$ denotes the block size in episode n. As a result, each episode processes $\lfloor \Delta N_{\text{episode}} / |b_n| \rfloor \times |b_n|$ transactions, including transactions invalidated in MVCC validation due to conflicts. Specifically, when the number of committed transactions *in consensus* reached T^n, AdaChain early aborts transactions in the ordering phase (i.e., no more transactions will be committed by the consensus protocol) until AdaChain transitions into the next episode (lines 16-17). In AdaChain, the block formation thread waits for transactions to be committed, cuts the block, possibly performs dependency graph construction, reordering, or execution according to the current architecture, and lastly, commits the block. Once the number of committed blocks reaches the high watermark, the block formation thread will be paused until action a_{n+1} is derived (lines 18-19). This ensures exactly $\lfloor \Delta N_{\text{episode}} / |b_n| \rfloor$ blocks are committed in episode n on different servers. Note that the learning phase (including feature extraction, exchanging measurements, training, and inference) is triggered by low watermark W_L^n, and AdaChain keeps processing transactions using architecture a_n between W_L^n and W_H^n. Thus, as shown in Section 7.5, architecture a_{n+1} is derived before reaching W_H^n in most cases, ensuring high throughput of the system.

5.2 Slow Path

Before AdaChain converges to the optimal architecture, the learning agent might occasionally choose "bad" architectures. The bad architectures might result in a high fraction of transactions being invalidated, or a slow growth of committed blocks (e.g., choosing OX when the workload is highly compute-intensive, or choosing XOV+reorder when the contention is extremely high). In terms of wall-clock time, AdaChain should not be stuck in either scenario. While the normal path is capable of handling the first scenario, we further introduce a slow path to handle the scenario where the growth of committed blocks is slow.

Algorithm 2 presents the slow path operations. When server i timeouts and the index of the last committed block, b_{last}, has not reached the low watermark, server i pauses the block formation after committing the current block, records the performance p_i^n in the current episode, and multicasts a s-checkpoint message including

the b_{last} to all servers (lines 1-4). If a server j receives s-checkpoint messages from at least $f + 1$ servers, even if its timer has not expired, it pauses the block formation, records its performance, and multicasts a s-checkpoint message to all servers (lines 5-8).

When the leader receives s-checkpoint messages from a quorum of $2f + 1$ servers, it finds the maximum index of the last committed block across all servers, W_H^n, and multicasts a s-propose message including W_H^n and the received $2f + 1$ s-checkpoint messages to all servers (lines 9-11). All servers validate the received s-propose message before two rounds of s-accept and s-commit communication, as shown in lines 12-16 (similar to the normal path). Each server then uses W_H^n as its high watermark and then resumes the block formation thread (lines 17-18). This ensures that in a slow path, the same number of blocks are committed in episode n across different servers. These operations might be expensive on the normal path, but are negligible on the slow path, compared to the timeout (15s in our case) and the poor performance before timeouts. The worst case happens when a fast server has not sent a s-checkpoint message, or its message has not been considered in the leader's calculation of W_H^n. In this case, if the index of its last committed block is higher than W_H^n, the server needs to rollback those exceeding blocks. Similar to the normal path, each server also needs to exchange state and performance measurements to derive action a_{n+1} for the next episode (lines 19-20).

Upon receiving transaction t for ordering at the leader l, the leader tags t with the current episode n as part of the sequence number (lines 21-22). When a server receives transactions committed by consensus protocol, it aborts transactions whose tagged episode is not equal to the current episode (lines 23-25). This ensures episode independence, i.e., there are no pending blocks across episodes in AdaChain. As a result, a bad architecture that triggers the slow path will not affect the performance of future episodes with promising architectures.

The normal path and slow path of AdaChain ensure two properties. First, transactions are strong serializable, and second, the world state is eventually consistent across different servers.

6 Security Analysis

Compared to fixed-architecture blockchains, our use of machine learning and run-time architecture switching add new security risks. In this section, we briefly present the new threats and discuss how AdaChain addresses them.

Adversarial ML. As studied in the ML community, machine learning can be adversarial [34, 45, 49, 63]. In the context of AdaChain, using reinforcement learning does not affect the *correctness* of the system, which depends only on the consensus protocol (predefined in our system), the current architecture, and our switching protocol. That being said, AdaChain's correctness guarantee is always as strong as the weakest architecture in its action space, regardless of the specific reinforcement learning algorithm.

However, reinforcement learning introduces a new *performance* attack vector: manipulating feature data to cause the learning agent to pick a bad architecture. To carry out this attack, a malicious node might propose adversarial feature values into the quorum Q in Algorithm 1. There are at least two ways such adversarial features could negatively impact performance: (1) decision attacks that target the inference phase, where an adversary reports false observations of its own features in order to push the global feature in one direction or another; and (2) poisoning attacks that target the training phase [25], where an adversary reports carefully selected feature values and labels to cause the next trained model to be inaccurate.

AdaChain mitigates such attacks by selecting the median value of all reported features. For a feature with low variance, adversaries would only be able to move the median value by a small amount even if they can report strong outliers (e.g., infinity or zero). However, for a feature with high variance, adversaries could potentially create a non-trivial change in the median value, impacting future inference and training. Since AdaChain is designed to operate in a homogeneous environment, most existing features and labels have low variance.

Thus, while adversarial ML attacks cannot impact the correctness of AdaChain, there are open questions about potential performance attacks caused by learning. Studying the full impact of adversarial learning on a system like AdaChain is an interesting avenue for future work.

Choosing different architectures. If peers choose different architectures within the same episode, the world state across peers can diverge and lead to correctness issues. AdaChain guarantees that every honest node agrees on the same new architecture in the same episode. We provide an analysis as follows.

Each node's learning agent starts with the same random seed when it is launched. Thus, since both the state and reward of a certain episode are the same across all honest nodes (as mentioned in Section 5), with the predictive model's deterministic training and inference, each honest agent chooses the same blockchain architecture in the same episode. Moreover, although dishonest nodes are tempted to make decisions that are different from honest nodes, they cannot affect the agreement on architecture among honest nodes. Without loss of generality, we assume $f = 1$ and there are $3f + 1$ peers in the system, where P_1, P_2, P_3 are honest and P_4 is malicious; we further assume the honest peers choose the XOV architecture.

Case 1: P_4 is the leader in the consensus protocol. There are four possible scenarios, none of which poses a correctness issue: (1) if P_4 chooses OX in the same episode, it will send out transaction proposals instead of endorsements in the ordering phase, so honest peers will detect this mismatch and initiate a view-change; (2) if P_4 chooses XOV but with a batch size b_4 that is different from what honest learning agents have chosen, it does not affect the agreement; (3) if P_4 chooses XOV with different batch sizes and uses different batch sizes for different peers (e.g., $b_{4,1}$ for P_1, $b_{4,2}$ for P_2, etc.), the honest peers will detect and reject these batches during consensus on batches and initiate a view-change; and (4) finally, if P_4 chooses XOV but with an opposite reordering choice, since reordering happens locally on each peer according to its local model, the malicious peer cannot corrupt honest peers.

Case 2: P_4 is not the leader in the consensus protocol. The honest leader detects type mismatch for messages originating from P_4 and discards them, while other honest peers work as normal.

Delay in architecture switching. An adversary might deliberately delay its own communication (sending messages) during architecture switching. In this scenario, if the adversary is a backup node

in the switching protocol, it does not hurt the system's throughput assuming the number of faulty backups is less than f. On the other hand, if the adversary is the leader, it could carry out a performance attack by delaying the transition into the new architecture, where the delay is carefully chosen to avoid triggering the timeout. Fortunately, there are known orthogonal techniques to mitigate these attacks. For example, instead of using a pessimistic switching protocol (inspired by PBFT), we can adopt a robust switching protocol, e.g., following Prime [11] which is a BFT consensus protocol robust to performance attacks.

In the partial synchrony model used by AdaChain, an adversary cannot indefinitely delay honest nodes. This is because in the partially synchrony model, there exists a global stabilization time (GST), after which all messages between honest nodes are received within some bound Δ. Even if we remove this partial synchrony assumption, correctness is still guaranteed at the expense of some performance degradation. It should also be noted that such attacks are not specific to AdaChain. In a fixed blockchain architecture, such timing-based attacks are also possible (e.g. adversaries delay messages in the ordering phase), and AdaChain does not exacerbate this attack.

Other vanilla threats. Other common Byzantine failures might also occur during the normal path and slow path operations. For instance, a malicious leader could send different propose messages to different backups, or forge a deviated global state and reward. In the meantime, a malicious backup node could double vote. AdaChain handles these threats using a PBFT-style switching protocol, which guarantees that the committed state and reward of a certain episode is the same across all honest nodes. We refer readers to the original paper [20] for more details.

7 Evaluation

Our evaluation aims to answer the following questions:

(1) Can AdaChain converge to the optimal architecture under a static workload without prior knowledge? (Section 7.2)

(2) How well does AdaChain perform compared to existing fixed blockchain architectures when the workload changes? (Section 7.3)

(3) How does the hardware setup (e.g., the type of CPU, network latency and bandwidth, etc.) affect the performance of AdaChain and existing blockchain architectures? (Section 7.4)

(4) What overhead does AdaChain introduce? (Section 7.5)

7.1 Experimental Setup

We have implemented a prototype of AdaChain in C++ and Python. The blockchain peers which process transactions and carry out architecture switching are implemented in C++. We use gRPC for communications between peers and LevelDB [1] for storing the world states. The learning agents are implemented separately in Python due to its mature machine learning libraries. Each peer communicates with its local learning agent through gRPC.

Testbed. Our testbed consists of 4 c6220 bare-metal machines on CloudLab [30], each with two Xeon E5-2650v2 processors (8 cores each, 2.6Ghz), 64GB RAM(8 x 8GB DDR-3 RDIMMs, 1.86Ghz) and two 1TB SATA 3.5" 7.2K rpm hard drives. These machines are connected by two networks, each with one interface: (1) a 1 Gbps Ethernet control network; (2) a 10 Gbps Ethernet commodity fabric. Unless otherwise specified, we use the second network for all communication. We set the size of the execution thread pool equal to the number of cores on each peer.

Workload	P_w	P_{hot}	N_{hot}	N_{trans}	T_{fire}	$T_{compute}$
A	0.2	0.95	5	300	50ms	5ms
B	0.5	0.99	10	100	50ms	1ms
C	0.5	0.1	10	300	50ms	10ms
D	0.9	0.95	1	100	50ms	0ms
E	0.5	0.99	10	100	50ms	5ms

Table 3: Specific workload parameters.

System configuration. We run a single blockchain channel consisting of 3 peers on 3 different servers. As mentioned in Section 3, each peer in AdaChain serves as an executor as well as an orderer. The choice of consensus protocol is configurable inside AdaChain, and we use Raft [53] for consistency with Hyperledger Fabric and its variants. We run the client on a separate server with 3 threads, each firing transaction proposals to one specific peer. The reported throughput only considers *effective* transactions, i.e., excluding early aborted and invalidated transactions. Throughout this paper, we parameterize the architecture switching protocol as follows: normal path timeout is set to 15s, the low watermark is set to 7500 transactions, and the high watermark to set to 10000 transactions.

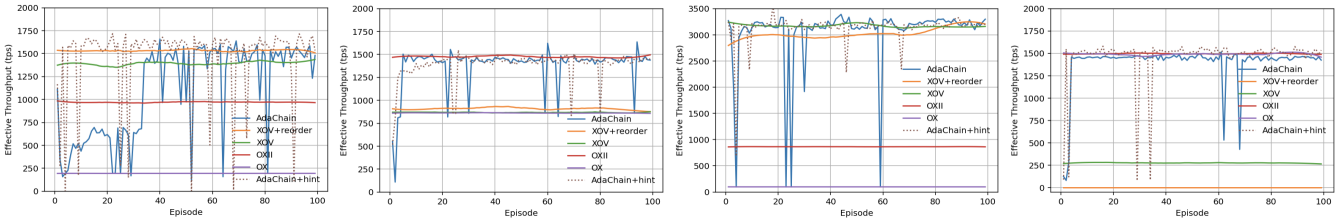
Workloads. To capture the diversity in real-world blockchain transactions, we implement a benchmark driver above SmallBank [29] to derive customized workloads with tunable parameters. The benchmark driver preloads the blockchain with 10k users, each with two accounts. We set N_{hot} of them as hot accounts. When firing transactions, the client randomly picks one of the five modifying transactions with probability P_w and the read-only transactions with probability $1 - P_w$. Each transaction has a certain probability to access the hot accounts, as controlled by the P_{hot} parameter. The client continuously fires N_{trans} transactions every T_{fire} milliseconds. To simulate computation-heavy transactions, each transaction has a $T_{compute}$ interval after it fetches the required world states from the key-value store and before its subsequent operations.

We use workloads `A-E` throughout this paper, where workloads `A-D` are the same as in Figure 1. The specific parameters of workloads `A-E` are listed in Table 3. Unlike workloads `A-D`, the additional workload `E` is introduced later in the section that explores adaptivity to different hardware settings. Although workload `E` has only slight deviation from workload `B`, it renders blockchain architectures extremely sensitive to hardware setup (details in Section 7.4). Note that we have written our own benchmark driver because no existing benchmark captures all these variations in workloads.

7.2 Convergence under Static Workloads

Our first set of experiments aims to demonstrate that *AdaChain can rapidly converge to the optimal architecture under a static workload with no prior experiences required.* We run AdaChain for 100 episodes on four representative workloads (i.e., `A-D`). To compare AdaChain against the *a priori* optimal architecture, we also perform a grid search in the action space to find the optimal architecture for each workload. For our four workloads, we compare AdaChain with the four optimal static architectures.

Figure 3 plots the performance of AdaChain and four baselines for each workload, where the baseline curves are smoothed for better readability. The curve for AdaChain is not smoothed. Each workload has a different optimal architecture. For instance, XOV+reorder



(a) Workload A (b) Workload B (c) Workload C (d) Workload D

Figure 3: Convergence of AdaChain to the optimal architecture under static workloads. Each plot shows the performance of AdaChain (w/ and w/o hint), XOV+reorder, XOV, OXII, and OX. While different blockchain architectures are optimal for each workload, AdaChain always reaches near-optimal performance.

Workload	Effective Throughput					AdaChain's Conv. Time	
	XOV+reorder	XOV	OXII	OX	AdaChain	w/ hint	w/o hint
A	**1532**	1415	968	194	1425	0.65	2.48
B	897	866	**1545**	861	1426	0.42	0.62
C	3228	**3235**	940	98	3153	0.45	0.48
D	1	272	1494	**1498**	1447	0.43	0.43
Average	1414	1447	1237	663	**1862**	0.49	1.00
Worst	1	272	940	98	**1425**	0.65	2.48

Table 4: Effective throughput (tps) for each architecture in the last 20 episodes of each workload and the convergence time (minutes) of AdaChain.

is optimal for workload A, suboptimal for workloads B and C, but the worst for workload D. Unlike a fixed architecture that cannot adjust itself even under a static workload, AdaChain always converges to the optimal architecture for each workload quickly within 40 episodes, no matter how bad the first episode (initial architecture) is. Due to Thompson sampling, AdaChain still performs some exploration in the architecture space even after convergence, as identified by the drops in the performance plot. Although these explorations do not seem useful under static workloads, they are crucial for finding optimal architectures under a changing workload which is more realistic in today's BaaS environment.

Comparing to exhaustive grid search (not shown in Figure 3) that takes n_a (the size of action space, i.e., 100 in our case) to converge and performs pure random exploration at all times, AdaChain converges much faster and strikes a better balance between exploiting known good actions and exploring unknown actions. Table 4 shows AdaChain's typical convergence time. In our definition, convergence is reached when staying within 95% of the optimal performance for 5 consecutive episodes.

Average performance. AdaChain obviously does not outperform the optimal action in any workload (it is the optimal action, after all). However, we show that AdaChain does offer good average and worst-case performance after convergence. Table 4 shows the throughput for each blockchain on each workload in the last 20 episodes of execution. Even with a few performance drops due to exploration, AdaChain achieves both the best average throughput *and* the best worst-case throughput across all four workloads.

Providing hints. An experienced administrator can build his knowledge into the learning framework by specifying certain rules in order to prevent some sub-optimal decisions. An example hint is "if the compute intensity is higher than 2000 us, enable early execution; otherwise, disable early execution". Hints of this nature

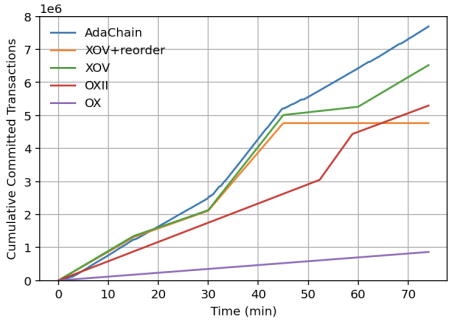


Figure 4: Cumulative committed transactions with time for a changing workload, showing AdaChain's ability to maintain superior performance during workload shifts.

reduce the search space, thus providing faster convergence and avoid certain explorations. Figure 3 summarizes our findings using the example hint (see "AdaChain+hint" lines). For workload A, the hint accelerates the convergence time of AdaChain by 3.8×. However, it is not sufficient to avoid all explorations in AdaChain, i.e., varying reordering choice and block size. In contrast, for workloads B and C where the convergence is already fast, the additional hint reduces unnecessary explorations after convergence.

7.3 Adaptivity under a Changing Workload

Our next experiment focuses on the key benefits of AdaChain: *when the workload is changing, AdaChain can commit significantly more transactions than the best baseline during the same deployment period.* To emulate a changing workload, we run workload A for the first 15 minutes, followed by workloads B, C, D, and A, each for 15 minutes. We use the same four baselines as in Figure 3, which are the optimal architectures for workload A-D when they are static.

Figure 4 shows the number of cumulative committed transactions with respect to time. During the entire 75 minutes, AdaChain successfully completed 7.73×10^6 committed transactions, while the best baseline XOV completed 6.60×10^6 committed transactions. The worst baseline OX only completed 0.87×10^6 committed transactions. AdaChain can successfully commit 1.12 million (17%) more transactions than the best baseline during 75 minutes. The trend in Figure 4 also suggests the improvement of AdaChain would become increasingly significant with a longer deployment time and more variations in the workloads, which are common in today's BaaS.

Interestingly, Figure 4 also shows a "catastrophic" effect for certain fixed architectures when the workload is changing. For instance, when transitioning back to workload A again (60-75 min),

the slope of XOV+reorder is near zero, indicating poor performance where few if any transactions are completed successfully. However, if we start running XOV+reorder right from the beginning under workload A without any changes to the workload (Figure 3(a)), XOV+reorder would be the optimal architecture. XOV+reorder performs poorly in workload D (45-60 min) due to the high overhead of Johnson's algorithm with a large number of cycles, which slows down the block formation. Since the block formation is sequential, the number of pending blocks grows significantly. Thus, incoming transactions simulate on stale data and would fail in the MVCC validation phase, even when transitioning back to workload A again. OX suffers from similar problems due to a large number of pending blocks. This phenomenon also justifies our watermark-based design of AdaChain, as elaborated in Section 5.

To further investigate how AdaChain switches its architecture under a changing workload, we plot AdaChain's effective throughput in each episode during the 75 minutes in Figure 5. The red dashed vertical line indicates when the workload shifts. Although workloads A and B have the same duration in terms of wall clock time (15 min), they vary in terms of the number of episodes. This is because, depending on the transaction arrival rate and compute intensity of different workloads, each episode (which is marked by the high watermark) may have a different time duration.

Figure 5 shows that when workloads shifts, AdaChain is able to quickly converge and perform competitively with the optimal architecture. For instance, when transitioning from workload A to B, AdaChain quickly converges to the new optimal (i.e., OXII) and achieves a 1450 tps throughput. In contrast, while XOV+reorder is optimal under workload A, as shown in Figure 3(b), it is able to reach only 900 tps when processing workload B, even in the best-case scenario where the catastrophic effect is avoided by starting with workload B and XOV+reorder right at the beginning. When transitioning from workload B to C, AdaChain quickly converges to the new optimal (i.e., XOV) and achieves a 3250 tps throughput. In comparison, OXII, which is optimal under workload B, is able to achieve only 980 tps under workload C (Figure 3(c)).

Due to Thompson sampling, AdaChain maintains some degree of exploration in the architecture space even after convergence, so as to avoid getting stuck at the local optimum. As AdaChain gains more experiences (i.e., data points) on a certain workload, the extent of exploration decreases, which is indicated by the less frequent drops within each 15 minutes period. Also, when AdaChain encounters a workload it has seen before (e.g., transition to workload A again in the last 15 minutes), AdaChain converges much faster than the first time and has less variation in performance.

Ideally, since we use the state of the previous episode to approximate the state of the next episode, AdaChain can adapt to workloads that shift less frequently than every two episodes (20s at most). In practice, due to the exploration performed by CMAB, as long as the workload changes are slower than the convergence time (as shown in Table 4), AdaChain can still operate effectively.

7.4 Adaptivity under Different Hardware

Our next set of experiments demonstrates another operational benefit of AdaChain: *when deployed on different hardware configurations, AdaChain can rapidly converge to the optimal architecture for that hardware without manually re-configuring the blockchain architecture.* We use workload E on three different hardware setups: HW1 stands for single data center deployment, where the network connecting peers has low latency (0.15 ms) and high bandwidth (10 Gbps), and each peer has 16 CPU cores; HW2 also stands for single data center deployment, but with 2 CPU cores per peer; HW3 stands for a multi-data center deployment, with high latency (50 ms) and low bandwidth (1 Gbps) network, and 2 CPU cores per peer. As mentioned in Section 7.1, for HW1 and HW2, we use the commodity network fabric; for HW3, we use the control network as well as Linux `netem` [32] to inject delay to the NIC. For each hardware setup, we also perform a grid search to find the optimal architecture for that hardware: for HW1, the optimal is (OXII, blocksize = 100); for HW2, the optimal is (XOV, blocksize = 1), for HW3 the optimal is (OXII, blocksize = 100) again. Figure 6 plots AdaChain's performance in each episode on HW1-HW3, along with the averaged performance of the optimal architecture for comparison.

As shown in Figure 6, even under the same workload, the hardware setup affects the effective throughput and thus affects the choice of the best architecture. For instance, OXII performs well when each server has enough compute resources (HW1 best arch in Figure 6(a)), but suffers when the servers have low compute resources (HW1 best arch in Figure 6(b)); StreamChain can perform well in a single data center deployment (HW2 best arch in Figure 6(b)), but suffers from the high round trip time when deployed across multiple data centers due to the small batch size it uses in the consensus protocol (HW2 best arch in Figure 6(c)). No matter what type of hardware AdaChain is deployed on, AdaChain can adapt itself to the optimal architecture for that hardware.

More importantly, for any kind of unseen hardware setup, users of AdaChain do not need to recollect data and retrain the machine learning model offline. AdaChain is an online system that learns from its past experiences and balances exploitation and exploration. With AdaChain, BaaS can have humans completely out of the loop.

	featurization	communication	training	inference	episode
mean	$0.11s \pm 0.01$	$0.14s \pm 0.04$	$0.21s \pm 0.04$	$0.01s \pm 0.01$	$3.67s \pm 2.12$
median	0.11s	0.14s	0.21s	0.01s	2.57s
max	0.20s	0.27s	0.32s	0.02s	16.35s

Table 5: Overhead of each stage in AdaChain.

7.5 Overhead of Learning

Our last experiment evaluates the additional overhead incurred by AdaChain's learning framework. We repeat the experiment in Section 7.3 and profile every stage that involves the learning agent. We report the results along with the episode duration in Table 5.

Before deriving the architecture for the next episode, AdaChain needs to go through feature extraction, communication, training and inference phases in sequence (Section 3). Table 5 shows that all of these four phases have low average overhead with low variance, as compared to the episode duration. The additional mean overhead time (0.47s) is only 12.8% of the average episode duration (3.67s).

More importantly, the 12.8% overhead can be masked by parallelizing transaction processing and learning. The learning phase only occurs between the low and high watermark period, which constitutes 25% duration with each episode. Considering that the median episode duration is 2.57s, this interval time is 0.64s which is higher than the mean overhead of 0.47s. During this interval,

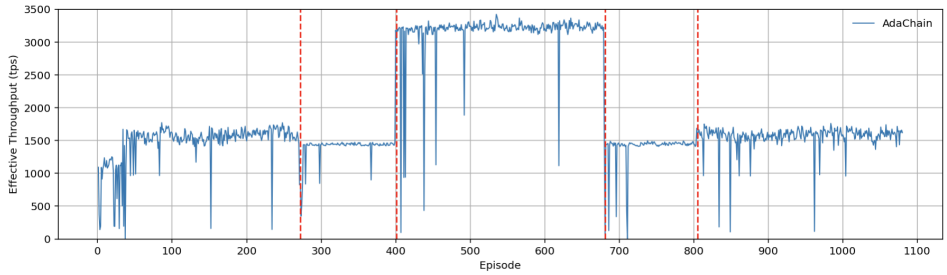
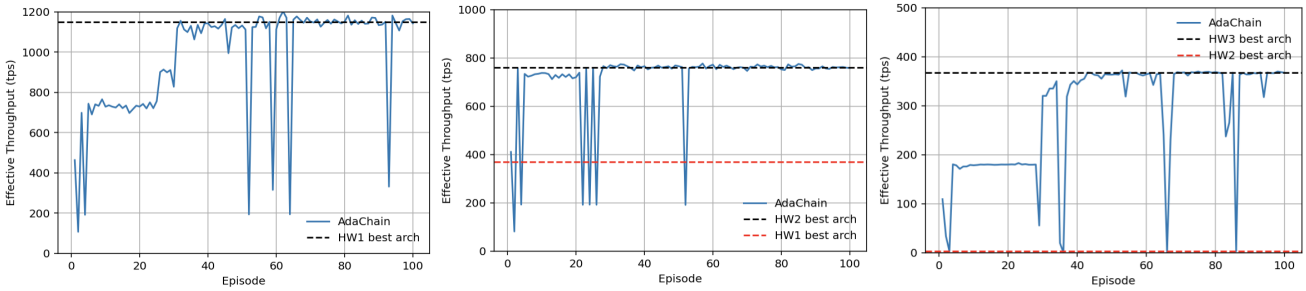


Figure 5: Effective throughput of AdaChain in each episode. Here, red vertical line indicates when the workload shifts. The workload shifts every 15 minutes. The number of episodes per 15 minutes duration varies depending on the transactions arrival rate and compute intensity of workload.



(a) HW1 (16 cores, RTT=0.15ms, BW=10Gbps) (b) HW2 (2 cores, RTT=0.15ms, BW=10Gbps) (c) HW3 (2 cores, RTT=50ms, BW=1Gbps)

Figure 6: Convergence of AdaChain to the optimal architecture under different hardware.

the peers continue to process transactions based on the current architecture, while in parallel, the learning agent goes through the four stages to determine the architecture for the next episode. Such parallel execution ensures that the overhead of AdaChain does not adversely affect its effective throughput, as long as there are some spare CPU cycles devoted to the learning agent.

Finally, we note that the episode duration has a high standard deviation due to two reasons: (1) AdaChain triggers the slow path when making poor architecture choices; (2) even in the normal path, the wall-clock time needed to reach a high watermark depends on the workload, e.g., the episode duration is longer under a compute-intensive or low-load workload.

Unlike deep neural networks, which are especially expensive to train, the random forest model used by AdaChain has moderate training overhead. With thousands of data points in the experience buffer, AdaChain only incurs a maximum training overhead of 0.32 seconds. Moreover, the succinct action space design also results in a lightweight inference phase, i.e., 0.02 seconds. When AdaChain is deployed for a long run, techniques such as periodic resampling and limiting the length of experience buffer [51] can be utilized.

8 Related Work

Learned systems. More generally, many recent works have applied machine learning concepts to systems components. These works, falling under the umbrella of machine programming [35], cannot be exhaustively enumerated here, but we refer the reader to past work on indexing [44], cardinality estimation [43, 48], index selection [28], database tuning [55], scheduling [50], garbage collection [21], and concurrency control for in-memory databases [62]. As a novel application, learned permissioned blockchains not only require unique featurization of the blockchain design landscape (i.e., action space), but also operate in an environment where there are Byzantine failures. Consequently, our design uses a fully decentralized machine learning approach to handle the untrustworthiness of participating nodes.

Database migration. Efficient and live migration of databases has been studied in multi-tenant data infrastructures [19, 27, 31, 40, 47]. Unlike existing work that mainly migrates data between different physical nodes, AdaChain switches between system architectures within the same participating node. Moreover, AdaChain's migration protocol is robust to Byzantine failures and mitigates the explorations performed by reinforcement learning.

9 Conclusion

In this paper, we presented AdaChain, an adaptive blockchain framework that leverages reinforcement learning to dynamically switch between different blockchain architectures based on the workload. AdaChain is able to identify the optimal blockchain architecture as workload changes, obtaining significantly higher throughput compared to fixed architectures. The low additional overhead of AdaChain can also be masked by parallelizing the transaction processing and learning phases. As future work, we are exploring expanding the learning framework to cover other aspects of blockchain architectures, e.g., choosing the best performing consensus protocol. Another intriguing future direction is to figure out whether our learning framework can be used to uncover new effective architectures not previously explored by human experts.

References

[1] [n. d.]. https://github.com/google/leveldb

[2] [n. d.]. AWS. Amazon quantum ledger database (QLDB). https://aws.amazon.com/qldb/.

[3] [n. d.]. Blockchain on AWS Enterprise blockchain made real. https://aws.amazon.com/blockchain/.

[4] [n. d.]. Chain. http://chain.com.

[5] [n. d.]. Corda. https://github.com/corda/corda.

[6] [n. d.]. Hyperledger Caliper. https://www.hyperledger.org/use/caliper.

[7] [n. d.]. Hyperledger Iroha. https://github.com/hyperledger/iroha.

[8] [n. d.]. IBM Blockchain Platform. https://www.ibm.com/cloud/blockchain-platform.

[9] [n. d.]. Oracle Blockchain. https://www.oracle.com/blockchain/.

[10] Shipra Agrawal and Navin Goyal. 2013. Further Optimal Regret Bounds for Thompson Sampling. In *The International Conference on Artificial Intelligence and Statistics (AISTATS '13)*.

[11] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computing* 8, 4 (2011), 564–577.

[12] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. In *Int. Conf. on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.

[13] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *SIGMOD Int. Conf. on Management of Data*. ACM, 76–88.

[14] Mohammad Javad Amiri, Joris Duguépéroux, Tristan Allard, Divyakant Agrawal, and Amr El Abbadi. 2021. SEPAR: Towards Regulating Future of Work Multi-Platform Crowdworking Environments with Privacy Guarantees. In *Proceedings of The Web Conf. (WWW)*. 1891–1903.

[15] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. 2022. Qanaat: A Scalable Multi-Enterprise Permissioned Blockchain System with Confidentiality Guarantees. *Proc. of the VLDB Endowment* 15, 11 (2022), 2839–2852.

[16] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*. ACM, 30:1–30:15.

[17] Leo Breiman. 1996. Bagging Predictors. In *Machine Learning (Maching Learning '96)*.

[18] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (Oct. 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[19] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2859–2872.

[20] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, Vol. 99. USENIX Association, 173–186.

[21] Lujing Cen, Ryan Marcus, Hongzi Mao, Justin Gottschlich, Mohammad Alizadeh, and Tim Kraska. 2020. Learned Garbage Collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL @ PLDI '20)*. ACM. https://doi.org/10.1145/3394450.3397469

[22] Jeeta Ann Chacko, Ruben Mayer, and Hans-Arno Jacobsen. 2021. Why Do My Blockchain Transactions Fail? A Study of Hyperledger Fabric. In *SIGMOD Int. Conf. on Management of Data*. ACM, 221–234.

[23] Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of Thompson sampling. In *Advances in neural information processing systems (NIPS'11)*.

[24] JP Morgan Chase. 2016. Quorum white paper.

[25] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. [n. d.]. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. arXiv:1712.05526 [cs] http://arxiv.org/abs/1712.05526 type: article.

[26] Sam Daley. 2021. 18 Blockchain-as-a-Service Companies Making the DLT More Accessible. https://builtin.com/blockchain/blockchain-as-a-service-companies.

[27] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.

[28] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *38th ACM Special Interest Group in Data Management (SIGMOD '19)*.

[29] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *SIGMOD Int. Conf. on Management of Data*. ACM, 1085–1100.

[30] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of {CloudLab}. In *Annual Technical Conf. (ATC)*. USENIX Association, 1–14.

[31] Aaron J Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 301–312.

[32] The Linux Fundation. 2021. netem. Retrieved July 4, 2022 from https://wiki.linuxfoundation.org/networking/netem

[33] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. 2022. Hybrid blockchain database systems: design and performance. *Proceedings of the VLDB Endowment* 15, 5 (2022), 1092–1104.

[34] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[35] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B. Tenenbaum, and Tim Mattson. 2018. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. Association for Computing Machinery, Philadelphia, PA, USA, 69–80. https://doi.org/10.1145/3211346.3211355

[36] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. 2022. *Diablo-v2: A Benchmark for Blockchain Systems*. Technical Report.

[37] Gideon Greenspan. 2015. MultiChain private blockchain-White paper. *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf* (2015).

[38] Hyperledger. [n. d.]. Private Data Collections: A High-Level Overview. https://www.hyperledger.org/blog/2018/10/23/private-data-collections-a-high-level-overview.

[39] Zsolt István, Alessandro Sorniotti, and Marko Vukolić. 2018. StreamChain: Do Blockchains Need Blocks?. In *Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL)*. ACM, 1–6.

[40] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. 2022. Remus: Efficient Live Migration for Distributed Databases with Snapshot Isolation. In *Proceedings of the 2022 International Conference on Management of Data*. 2232–2245.

[41] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, Mike Dahlin, et al. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers.. In *Symposium on Operating systems design and implementation (OSDI)*, Vol. 12. USENIX Association, 237–250.

[42] Emilie Kaufmann, Nathaniel Korda, and Rémi Munos. 2012. Thompson sampling: An asymptotically optimal finite-time analysis. In *International Conference on Algorithmic Learning Theory (ALT '12)*.

[43] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*. http://arxiv.org/abs/1809.00677

[44] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA. https://doi.org/10.1145/3183713.3196909

[45] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236* (2016).

[46] Jae Kwon. 2014. Tendermint: Consensus without mining. (2014).

[47] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. 2019. MgCrab: transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment* 12, 5 (2019), 597–610.

[48] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON '15)*. IBM Corp., Riverton, NJ, USA, 53–59. http://dl.acm.org/citation.cfm?id=2886444.2886453

[49] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations*.

[50] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2018. Learning Scheduling Algorithms for Data Processing Clusters. *arXiv:1810.01963 [cs, stat]* (2018). http://arxiv.org/abs/1810.01963 arXiv: 1810.01963.

[51] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. China. https://doi.org/10.1145/3448016.3452838 Award: 'best paper award'.

[52] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718.

[53] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm. In *Annual Technical Conf. (ATC)*. USENIX Association, 305–319.

[54] Ian Osband and Benjamin Van Roy. 2015. Bootstrapped Thompson Sampling and Deep Exploration. *arXiv:1507.00300 [cs, stat]* (July 2015). http://arxiv.org/abs/1507.00300

[55] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*.

[56] Zhe Peng, Cheng Xu, Haixin Wang, Jinbin Huang, Jianliang Xu, and Xiaowen Chu. 2021. P2B-Trace: Privacy-Preserving Blockchain-based Contact Tracing to Combat Pandemics. In *SIGMOD Int. Conf. on Management of Data*. 2389–2393.

[57] Zhe Peng, Jianliang Xu, Xiaowen Chu, Shang Gao, Yuan Yao, Rong Gu, and Yuzhe Tang. 2021. Vfchain: Enabling verifiable and auditable federated learning via blockchain systems. *IEEE Transactions on Network Science and Engineering* (2021).

[58] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-order-validate Blockchains. In *SIGMOD Int. Conf. on Management of Data*. ACM, 543–557.

[59] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Int. Conf. on Management of Data*. ACM, 105–122.

[60] Feng Tian. 2017. A supply chain traceability system for food safety based on HACCP, blockchain & Internet of things. In *Int. Conf. on service systems and service management (ICSSSM)*. IEEE, 1–6.

[61] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated {Key-Value} Stores. In *Annual Technical Conf. (ATC)*. USENIX Association, 33–48.

[62] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control.. In *OSDI*. 198–216.

[63] Eric Wong, Leslie Rice, and J Zico Kolter. 2019. Fast is better than free: Revisiting adversarial training. In *International Conference on Learning Representations*.

[64] Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. 2022. FlexChain: An Elastic Disaggregated Blockchain. *Proc. of the VLDB Endowment* 16, 01 (2022), 23–36.

[65] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. 2020. LedgerDB: a centralized ledger database for universal audit and verification. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3138–3151.

[66] Li Zhou. 2016. A Survey on Contextual Multi-armed Bandits. *arXiv:1508.03326 [cs]* (Feb. 2016). http://arxiv.org/abs/1508.03326