Pufferfish: DAG-Guided Speculative Execution for Low-Latency BFT SMR

Jianting Zhang
Purdue University

Alberto Sonnino
Mysten Labs / UCL

Lefteris Kokoris-Kogias
Mysten Labs

Aniket Kate
Mysten Labs / Purdue University

***Abstract*—Modern Byzantine fault-tolerant state machine replication (BFT SMR) systems increasingly decouple data dissemination from transaction ordering, but typically execute transactions only after consensus finalizes their order. This execution-after-consensus design serializes ordering and execution on the confirmation path, so transactions incur both costs even under normal network conditions. Decoupling enables dissemination to continue during ordering stalls, but amplifies the same bottleneck: disseminated transactions accumulate into an execution backlog that replicas must drain after ordering recovers.**

This paper presents Pufferfish, a BFT SMR design that uses *speculative* execution to move execution off the post-consensus critical path. Replicas speculatively execute transactions according to a predicted pre-commit order derived from a DAG-based consensus substrate. To make this work reusable, Pufferfish combines adaptive probabilistic speculation, which predicts the most likely final order from the local DAG view, with commit-aware snapshots, which preserve selected prefix states to reduce replay when speculation is wrong. In the common case, Pufferfish overlaps execution with consensus to reduce confirmation latency. When ordering is delayed, the same mechanism converts otherwise idle time into useful execution work, limits backlog growth, and shortens recovery. We implement Pufferfish and evaluate it in a geo-distributed AWS deployment across five regions. At 80,000 transactions per second (TPS) with 10 replicas, Pufferfish reduces p99 confirmation latency from 681 ms to 430 ms (36.9%) relative to an execution-after-consensus baseline. With 49 replicas at 25,000 TPS near the baseline's saturation point, Pufferfish achieves a p99 latency of 306 ms, compared with 724 ms for the baseline. Under induced consecutive ordering stalls, it reduces latency from 17.1 s to 9.8 s, a 42.7% decrease.

I. Introduction

Modern replicated transactional systems, such as ledgers, decentralized asset registries, and decentralized finance engines, are built on Byzantine fault-tolerant state machine replication (BFT SMR), where a consensus layer establishes a global transaction log and an execution engine applies transactions to replicated state. These systems typically employ a log-then-execute [1]–[3] (or execution-after-consensus (EAC)) architecture, sharing the same fundamental design point as deterministic databases [4], [5]: transaction execution starts only after the global order is fixed. Consequently, ordering and execution are serialized on the transaction-confirmation path: even when the network is well-behaved, a transaction pays both consensus latency and subsequent execution latency.

To improve throughput, a prominent trend in modern BFT SMR has been to decouple *data dissemination* from *transaction ordering* [6], leading to a rich line of decoupled protocols [6]–[15]. Decoupling allows dissemination to proceed concurrently with ordering, but most designs still defer execution until consensus. This leaves execution on the common-case critical path and can create post-consensus queuing as the offered load approaches the execution capacity.

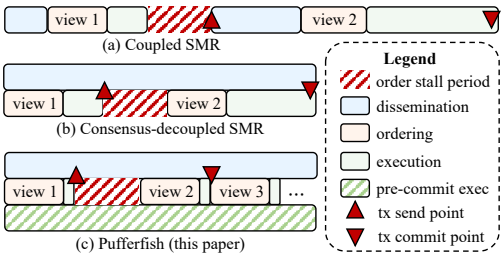


Fig. 1. Ordering stalls amplify execution-after-consensus (EAC) bottleneck. (a)(b) The EAC baseline defers execution and creates a recovery backlog; (c) Pufferfish executes against a predicted order before consensus recovers.

Ordering delays amplify this common-case serialization bottleneck, and such delays occur in deployed partially synchronous systems because of network turbulence, attacks, or implementation-level issues [15]–[22]. In a decoupled BFT SMR, disseminated transactions accumulate during ordering stalls, while execution waits for a finalized order. When ordering recovers, replicas face a burst of delayed execution during which replica commitment latency remains elevated until the backlog is drained. Our measurements under an induced ordering stall of 10 s show that confirmation latency remains degraded for an additional 7 s in the EAC baseline. We illustrate the amplified case in Fig. 1. In the EAC baseline, the engine sits idle during the stall, leading to a backlog spike and a long recovery period. This execution-side recovery bottleneck is not just a theoretical concern but has been observed in production as well: the Sui testnet experienced a prolonged disruption in June 2025, during which replicas processed a large backlog after ordering recovered (§ II-D).

This paper presents Pufferfish, a BFT SMR design that reduces confirmation latency through speculative execution. In Pufferfish, replicas speculatively execute transactions against a predicted order during the consensus. If the prediction matches the final consensus order, execution is already complete when ordering finishes, removing most execution work from the post-consensus critical path. This provides a common-case latency benefit by *overlapping consensus and execution*. During ordering delays, the same mechanism continues making execution progress, limiting the backlog that remains after

recovery, as illustrated in Fig. 1(c). In this way, Pufferfish transforms otherwise idle time during ordering stalls into useful execution work. Viewed through a database lens, this is optimistic concurrency control lifted to the ordering layer. Replicas execute transactions optimistically, validate against the finalized log, and repair state on mismatch. Importantly, safety is never at risk: speculative state is committed only after consensus confirms the order, so mis-speculation costs only performance, never serializability or linearizability.

The challenge is not speculative execution per se, but making speculative work *reusable*. Speculative work is beneficial only if it remains consistent with the eventual consensus order; otherwise, rollback and re-execution can negate its benefit or even introduce additional overhead. Prior speculative-execution BFT designs (such as HotStuff-1 [23] and Zaptos [24]) speculate directly on leader proposals, which may be replaced during view changes, invalidating speculative work across repeated stalls. Our evaluation of a proposal-based speculative system (§ VI-B) shows that repeated invalidation caused by view changes can even degrade performance. Therefore, Pufferfish must address two coupled challenges: (i) maintaining a speculative order that tracks the eventual consensus order across consecutive stalled rounds where ordering remains unstable for an extended period, and (ii) reducing the cost of mis-speculation by limiting rollback and re-execution, so that occasional prediction failures do not require replaying the entire speculative history.

Pufferfish addresses these challenges with three ideas (§ III).
DAG substrate. We build on Mysticeti consensus [9], whose directed acyclic graph (DAG) mempool exposes causal and voting information that replicas can exploit. In particular, Mysticeti preserves an append-only causal structure: newly disseminated blocks extend earlier causal histories rather than replacing them with independent proposals. The resulting preservation of previously implied order lets speculative work remain reusable across stalls when a later committed leader extends the predicted prefix.
Adaptive probabilistic speculation. To predict likely ordering outcomes, Pufferfish uses the local DAG structure to estimate leader-commit outcomes and continuously adjusts the speculative order toward the most likely final order, updating predictions incrementally as the DAG grows instead of restarting.
Commit-aware snapshots. Pufferfish takes selective, lightweight logical checkpoints of versioned prefix states through a reputation-guided [25], [26] placement strategy and a fixed-size recent-prefix window. When speculation fails, these checkpoints let replicas restore the latest available matching prefix and replay only the remaining suffix, keeping the repair cost manageable.

We implement Pufferfish on top of the Mysticeti codebase with a real EVM execution engine and evaluate it in a geo-distributed AWS deployment spanning five regions with up to 49 replicas. Compared with MysticetiEAC, an execution-after-consensus baseline using the same consensus substrate, Pufferfish reduces p99 confirmation latency from 681 ms to 430 ms at 80,000 transactions per second (TPS) with 10 replicas. With 49 replicas at 25,000 TPS near the baseline's saturation point, p99 latency falls from 724 ms to 306 ms. Under consecutive ordering stalls, it reduces latency from 17.1 s to 9.8 s. We also implement and evaluate Byzantine attacks designed to undermine Pufferfish's effectiveness, showing that Pufferfish is resilient to these attacks under both common and ordering-stall cases.

We summarize our contributions as follows:

- We identify and measure a consensus-execution pipeline bottleneck in decoupled BFT SMR (§ II): execution-after-consensus adds execution to the common-case critical path and turns ordering stalls into backlogs whose latency impact outlives the stall.
- We design Pufferfish (§ IV), which combines DAG-guided adaptive probabilistic speculation with commit-aware snapshots to make speculative execution reusable across both common and disrupted network conditions. To the best of our knowledge, this is the first work exploring effective speculative execution under consecutive ordering stalls for modern decoupled BFT SMRs.
- We implement Pufferfish and conduct extensive experiments (§ VI), demonstrating lower common-case latency and faster recovery from ordering stalls than several representative baselines, as well as the resilience to Byzantine attacks.

II. Background and Motivation

A. BFT SMR

Byzantine fault-tolerant state machine replication (BFT SMR) is the standard abstraction for building blockchains and other replicated services in the presence of arbitrary faults. In this model, clients submit transactions to a set of replicas, and the replicas must process those transactions so that all correct replicas observe a consistent committed history and reach consistent states.

Formally, let $St^i_{\vdash k}$ denote the state maintained by replica R_i after executing the first k committed transactions. A BFT SMR system must satisfy the following two properties:

- *Safety:* If two correct replicas R_i and R_j commit transaction sequences $(tx_1, \ldots, tx_m)$ and $(tx'_1, \ldots, tx'_{m'})$, then $tx_k = tx'_k$ and $St^i_{\vdash k} = St^j_{\vdash k}$ for every $k \leq \min\{m, m'\}$. That is, any two correct replicas commit the same prefix history and therefore maintain the same prefix states.
- *Liveness:* If a correct client submits a transaction to correct replicas, then all correct replicas eventually commit it.

The transaction-processing pipeline of a BFT SMR system can be viewed as three recurring tasks [27], [28]: *data dissemination*, which propagates transactions among replicas; *ordering*, which establishes a consistent committed order on disseminated transactions; and *execution*, which applies committed transactions to the replicated state. This decomposition exposes the bottleneck targeted in this paper: execution-after-consensus serializes ordering and execution on the common-case confirmation path, and any slowdown in ordering further delays the start of execution.

B. Partially Synchronous SMR in Practice

Most BFT SMR systems are designed under partial synchrony [29]. Informally, the model assumes that message delays may be unbounded for some time, but eventually become bounded by a delay Δ. This assumption is primarily a proof model for liveness: it allows a protocol to guarantee progress once the network behaves synchronously long enough.

Practical partially synchronous protocols realize this assumption through timeouts. Replicas wait for a bounded period before concluding that the current ordering attempt is not making progress, and then trigger recovery actions such as view changes, leader replacement, or fallback commit rules. In theory, timeouts are expected to stop firing once the network remains stable long enough. In deployed systems, however, network conditions may fluctuate repeatedly due to packet loss, transient congestion, targeted attacks, or implementation-level disruptions [15]–[22]. Recent work has formalized this behavior as network turbulence or intermittent instability in partially synchronous BFT deployments [15].

Ordering stalls. Following these works, we use an *ordering stall* to denote an interval in which the ordering layer cannot finalize transactions at its normal pace, often because timeout-driven recovery interrupts the optimistic ordering path. Stalls may occur *intermittently* in isolated rounds or *consecutively* across several rounds before a leader commits. An ordering stall does not imply a safety violation; it means only that the path for finalizing transaction order is temporarily delayed.

C. Decoupled BFT Consensus

Modern high-throughput BFT systems increasingly adopt *decoupled* consensus architectures, in which data dissemination is separated from the consensus process that finalizes transaction order. Decoupled consensus protocols can be broadly grouped into three families. *Batch-based* protocols [23], [28], [30]–[34] disseminate transactions in batches independently of ordering, and later order lightweight digests of those batches. *Multichain* protocols [7], [15] let each replica extend its own chain of disseminated blocks, while a separate ordering mechanism reasons over recent certified chain tips. *DAG-based* protocols [6], [8]–[14] organize disseminated blocks into a directed acyclic graph (DAG), where vertices carry transactions and edges encode causal references. Replicas later interpret the DAG to derive a committed order.

The common point across these families is not that they eliminate ordering disruptions, but that they allow data dissemination to continue with weaker dependence on the current ordering leader. As a result, decoupling mitigates the effect of ordering stalls on *data-movement throughput*. However, decoupling by itself does not allow replicas to execute transactions safely without a sufficiently stable committed order.

This distinction is crucial for our work. Prior work on network turbulence and decoupled consensus has primarily focused on the ordering path: how to maintain or recover progress when partially synchronous assumptions are repeatedly stressed [15]. In contrast, this paper focuses on the *execution consequences* of such ordering stalls. This also differentiates our setting from fully asynchronous BFT protocols [6], [35]–[37], which pursue liveness without relying on eventual synchrony, and from execute-order-validate architectures [38]–[42], which restructure the execution path differently. We discuss these alternatives in more detail in § VII.

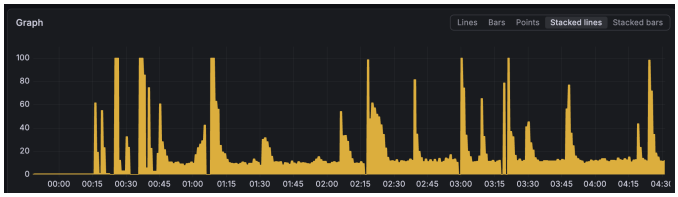


Fig. 2. Execution jitter during the Sui testnet incident (June 2025). The x-axis shows time (incident started at 00:15) and the y-axis shows transaction latency overhead (ms).

D. Motivation

Although decoupled consensus protects dissemination from temporary ordering stalls, it leaves an execution-layer bottleneck unresolved. In an execution-after-consensus pipeline, a transaction first waits for ordering and only then enters execution. Thus, even in the common case, its confirmation latency includes both stages serially. At a high offered load, this post-consensus execution stage can also introduce queuing directly on the confirmation path.

Ordering delays amplify the same bottleneck. Execution must respect a consistent finalized order to preserve replica safety, so replicas continue accumulating disseminated transactions but cannot execute them at the same pace while ordering is stalled. Once ordering recovers, they face a burst of delayed execution work. The latency impact therefore outlives the stall because replicas must drain this backlog before confirmation latency returns to normal. This is not merely a theoretical concern: the Sui testnet experienced a roughly one-day halt in June 2025 after replicas failed to locally confirm transactions and had to process a large backlog (Fig. 2) [43].

These observations motivate a broader design goal for Pufferfish: overlap execution with ordering whenever a sufficiently likely pre-commit order is available. This removes execution from much of the common-case critical path and, when ordering stalls, uses the additional delay to process work that would otherwise become a recovery backlog. The rest of the paper shows how to realize this overlap while preserving safety and mitigating the replay cost of incorrect speculation.

III. Model, Challenges, and Overview

This paper presents Pufferfish, a BFT SMR system that reduces confirmation latency by overlapping transaction execution with consensus. Its key idea is *speculative* execution, which speculatively executes transactions according to a predicted pre-commit order before consensus finalizes it. When predictions are correct, much of the execution work is completed before ordering finishes, reducing common-case latency and accelerating recovery from ordering stalls. When predictions are incorrect, replicas selectively repair the affected

state while preserving safety. This section presents the system model, design challenges, and an overview of Pufferfish.

Models. We consider $n{=}3f{+}1$ replicas $\mathcal{R} = \{R_1, R_2, \cdots, R_n\}$, of which up to f may be *Byzantine*. Byzantine replicas can behave arbitrarily but have bounded computational power and cannot break cryptographic primitives. The remaining replicas are *correct*. We assume a public-key infrastructure with digital signatures for authentication. As in prior BFT SMR systems [28], [44], [45], we assume partial synchrony [29], under which liveness is guaranteed only during periods of synchrony.

Challenges. Designing reusable speculative execution is non-trivial because execution is more order-sensitive than dissemination. While transactions can be disseminated without knowing their final positions, speculative execution results can be safely reused only if they remain consistent with the eventual consensus order.

This introduces two practical challenges. First, replicas must derive a speculative order early enough to overlap execution with consensus and keep it close to the eventual committed order, even *across consecutive stalled rounds*. Without sufficient pre-commit information, speculation becomes guesswork; without continuous refinement, prolonged ordering instability forces replicas to repeatedly discard and rebuild speculative work. Second, mis-speculation is unavoidable under asynchrony and Byzantine behavior. If each mistake requires discarding most speculative work and re-executing the backlog, speculation may lose its benefits or even perform worse than execution after ordering.

These challenges imply that Pufferfish needs more than a generic speculative executor. It needs: (i) a consensus substrate whose disseminated structure remains useful across consecutive stalled rounds, (ii) a speculation mechanism that turns that structure into a likely execution order and updates it online, and (iii) a rollback mechanism that preserves useful prefix states and reduces repair work when the prediction is wrong.

Overview. To meet these requirements, Pufferfish combines three components: (i) a DAG-based consensus substrate, (ii) an adaptive probabilistic speculation (APS) mechanism, and (iii) a commit-aware snapshot mechanism.

DAG-based consensus (§ IV-A). Pufferfish builds on Mysticeti [9], a representative DAG-based BFT consensus protocol. Mysticeti decouples dissemination from ordering, so transactions can continue propagating as replicas build the DAG while ordering temporarily stalls. As its DAG grows, it exposes causal and voting evidence before final ordering, allowing Pufferfish to begin execution during common-case consensus. The append-only causal structure shared by DAG-based consensus protocols preserves previously implied order when later committed leaders extend earlier causal histories, supporting consecutive speculation across several rounds. In the representative batch-based and multichain designs we examine, failed ordering proposals are replaced across rounds, making it harder to preserve and refine speculative execution state continuously. We defer the detailed comparison to Appendix A.

APS mechanism (§ IV-B). The DAG alone does not provide a final order, so replicas still need a mechanism that converts partial structural signals into a concrete speculative schedule. Pufferfish therefore introduces APS. APS analyzes the local DAG view, including the implicit evidence it carries about likely leader and commit outcomes, and derives the speculative order that is currently most likely to match the eventual committed order. As new blocks arrive, APS updates this prediction rather than restarting from scratch. In this way, APS starts useful execution before common-case finality and complements the DAG substrate in addressing the first challenge: it supports continuous speculation across evolving DAG views, maximizing the speculative work that remains reusable once ordering completes.

Commit-aware snapshot (§ IV-C). Even with APS, speculation can be wrong. Pufferfish therefore adds a commit-aware snapshot mechanism to reduce the frequency and extent of wholesale re-execution. Instead of eagerly snapshotting every speculative step, replicas retain a fixed-size window of recent prefix states and additional checkpoints around speculation points that appear more likely to be invalidated. When the final order diverges from the speculative one, a replica restores the latest available matching snapshot and re-executes the remaining suffix.

IV. Pufferfish Protocol

A. Mysticeti Consensus

Pufferfish uses Mysticeti [9], a DAG-based BFT consensus protocol. Mysticeti disseminates transactions in blocks while their references provide voting evidence for ordering. We summarize the relevant consensus rules.

DAG and leader decisions. In each round r, each replica broadcasts a block B_i^r containing transactions and references to at least $2f{+}1$ blocks from round $r{-}1$. After receiving $2f{+}1$ round r blocks, it can advance to round $r{+}1$; before producing a round-$(r{+}2)$ block, Mysticeti waits for $2f{+}1$ round-$(r{+}1)$ votes for the round-r leader or a timeout [9]. Blocks and references form an append-only DAG. Replicas keep extending their local DAG views even when an earlier leader decision stalls, exposing new evidence for subsequent rounds.

Each round has a designated leader block B_L^r. There are two patterns to decide B_L^r. A *certificate pattern* consists of at least $2f{+}1$ round $r{+}1$ blocks referencing B_L^r, while a *skip pattern* consists of at least $2f{+}1$ blocks that do not reference it. A later block whose causal history contains the certificate pattern is a *certificate* for B_L^r. Mysticeti directly decides B_L^r as *to-commit* if it has at least $2f{+}1$ certificates in round $r{+}2$, or as *to-skip* if it has a skip pattern. This is called Mysticeti's direct decision rule. If neither condition holds, the leader remains undecided. Its status is then determined by the first later leader (after round $r{+}2$) decided as to-commit: B_L^r commits if that leader's causal history contains a certificate for it, and is skipped otherwise. Thus, a leader with too few round $r{+}2$ certificates for direct commitment can still commit *indirectly* when a later committed leader includes one of its

certificates in its causal history, which is called Mysticeti's indirect decision rule.

Ordering and speculation opportunities. Once all leaders in a prefix are decided, Mysticeti processes them in round order. It omits skipped leaders and deterministically linearizes the still-unordered causal history of each committed leader [46]. The resulting block sequence gives replicas a total order for transaction execution.

Mysticeti exposes next-round evidence early enough to overlap common-case execution with consensus. Specifically, next-round references can indicate a leader's eventual status before consensus finalizes it. Moreover, since later committed leader blocks can preserve and extend the order that was already implied by earlier undecided leader blocks through causal history and the indirect decision rule [47], it makes multi-round speculative execution meaningful and increases the fraction of speculative work that can eventually be committed. The following subsection explains how Pufferfish uses this evidence to choose an execution order before commitment.

B. Adaptive Probabilistic Speculation

The purpose of APS is to maximize the amount of speculative work that can later be reused. Since transaction order in Mysticeti is determined by the sequence of leader blocks that are decided as to-commit, APS predicts an order for pending (i.e., unordered) leader blocks from the local DAG view and uses it to drive speculative execution. In particular, APS consists of a lightweight predictor that enables early execution and an APS tree that compactly updates the speculative order as the DAG advances, including across stalled rounds.

Predict statuses of leader blocks. Intuitively, the earlier a replica predicts, the sooner it can start execution, but the less evidence it has about the eventual ordering outcome. On the other hand, waiting longer improves confidence but also leaves more execution work on the critical path. The APS predictor balances this tradeoff with a *one-round deferred* policy: replica R_i predicts the status of a round r leader block once it has received a quorum of $2f+1$ blocks from the next round $r+1$. Fig. 3 summarizes this one-round deferred prediction process. Whenever R_i receives a quorum of blocks in round $r+1$, it predicts each as-yet-unpredicted leader block in rounds $r_{spl}+1$ through r (lines 4–8).

The prediction rule itself mirrors Mysticeti's decision logic, but in a simplified, best-effort form (lines 18–22). For a round r' leader block $B_L^{r'}$, if at least $2f+1$ blocks in round $r'+1$ directly reference $B_L^{r'}$ in R_i's local view, APS predicts to-commit; otherwise it predicts to-skip. The rationale is that direct support from the next round is a necessary precursor to eventual commitment under Mysticeti's direct and indirect decision rules. Thus, a leader block with $2f+1$ supporting references is a strong early indicator of a to-commit leader. Our ablation study (§ VI-D) validates its effectiveness: in the common case, APS achieves a 99.9% prediction hit rate; under different network jitter conditions and at varying replica scales, APS achieves an 88.1%–93.9% prediction hit rate, compared with 73.3%–83.3% for the best aggressive policy, and reduces p99 confirmation latency by 8.7%–28.7% (Fig. 9 and Table II).

Variables:
- $DAG_i[]$ - An array of sets of blocks (indexed by rounds)
- r_{spl} - The round of the last predicted leader block
- St_{spl} - The latest speculative state
- struct Leader Prediction LP:
  - $LP.block$ - the leader block
  - $LP.round$ - the round number of a pending leader block
  - $LP.status$ - the predicted status (to-commit/to-skip)
- $\mathcal{L}_{spl}$ - The APS tree, consisting of a list of LP

1: **upon** receiving a quorum of $2f+1$ blocks in round $r+1$ **do**
2: $sequence \leftarrow []$ ▷ newly predicted leader blocks
3: $LP \leftarrow ()$
4: **for** $r' \in [r$ down to $r_{spl}+1]$ **do**
5: $LP.block \leftarrow$ get_leader_block(r')
6: $LP.round \leftarrow r'$
7: $LP.status \leftarrow$ predict_status($DAG_i[r'+1], LP.block$)
8: $sequence \leftarrow (LP)||sequence$
9: $\mathcal{L}_{spl}$.append($sequence$)
10: $r_{spl} \leftarrow r$
11: $St_{spl} \leftarrow$ speculative_execution($St_{spl}, sequence$)
12: **upon** later round $r+1$ block with r pending **do**
13: $LP \leftarrow$ the pending round-r entry in $\mathcal{L}_{spl}$
14: $status \leftarrow$ predict_status($DAG_i[r+1], LP.block$)
15: **if** $LP.status =$ to-skip and $status =$ to-commit **then**
16: $LP.status \leftarrow$ to-commit
17: $St_{spl} \leftarrow$ refresh_speculation_from($r, \mathcal{L}_{spl}$) ▷ reuse prefix
18: **procedure** predict_status($blocks, B_L$)
19: **if** $B_L == \perp$ **then**
20: **return** to-skip
21: $voters \leftarrow \{B \in blocks : B_L \in B.parents\}$
22: **return** $|voters| \geq 2f+1$? to-commit : to-skip

Fig. 3. Leader prediction in APS for replica R_i.

Prediction correction. An initial quorum of $2f+1$ round $r+1$ blocks may contain f promptly delivered Byzantine non-votes and only $f+1$ honest votes for an honest round r leader B_L^r. APS then temporarily predicts to-skip because support falls below its $2f+1$ threshold.

APS corrects this initial bias by updating pending predictions as new blocks arrive. In particular, the remaining f honest round $r+1$ blocks raise supports/votes for B_L^r to $2f+1$, so APS changes its prediction to to-commit (lines 12–17). The replica then executes B_L^r from the existing prefix, replaying an affected suffix only if later leaders have already been executed (line 17). Correcting the prediction before speculation advances therefore avoids suffix replay. Mysticeti's leader-timeout mechanism allows time for this correction: before producing its round $r+2$ block, an honest replica waits until it observes $2f+1$ votes for B_L^r or the round timeout expires [9]. Since APS predicts the round $r+1$ leader only after receiving a quorum of round $r+2$ blocks, this wait allows the late honest votes to trigger correction under synchrony before speculation advances to the next leader. Our Byzantine evaluation (§ VI-C) demonstrates that APS correction effectively mitigates the impact of this Byzantine withholding behavior.

APS tree. APS organizes pending leader rounds in a local tree, with the earliest unresolved round at the root. Each level branches on whether its leader will commit or skip, so a root-to-leaf path represents a candidate leader sequence. The replica

follows its current predictions rather than enumerating every path. As consensus resolves earlier leaders, it prunes incompatible branches, retains the unresolved suffix, and extends the tree with newly pending rounds. This lets speculation continue across stalled rounds without rebuilding its entire order.

The selected leader sequence is passed to speculative_execution (line 11), which serializes and executes the causal histories of predicted to-commit leaders (Fig. 5, lines 45–58). If a leader block is missing, APS temporarily predicts to-skip so later rounds can proceed; the recovery mechanism in § IV-C repairs any resulting mis-speculation. APS does not alter consensus decisions or guarantee that its predictions match them.

C. Commit-aware Snapshots

Speculative's speculative execution may diverge from the eventual consensus order due to mispredictions. For example, network delays may cause an incomplete local DAG view, while Byzantine replicas may manipulate observations, causing a leader block eventually decided as to-commit to be predicted as to-skip. Such mispredictions require replicas to roll back and re-execute affected transactions. However, restarting from the last committed state after every misprediction could negate the benefits of speculation.

A straightforward solution is to retain a recovery snapshot after *every* speculative step, providing fine-grained rollback points. However, this incurs growing storage costs during long speculative runs. Instead, Pufferfish employs *commit-aware snapshots*, combining two complementary mechanisms (Fig. 4): (i) *reputation snapshots*, which provide long-lived checkpoints around speculative steps more likely to be invalidated, and (ii) a bounded *snapshot window*, which retains the most recent s_w speculative prefix states.

Reputation snapshots. Inspired by prior reputation-guided strategies [12], [25], [26], each replica R_i in Pufferfish maintains a local reputation table TR_i, where $TR_i[j]$ reflects the historical accuracy of predictions for leader blocks created by replica R_j. After consensus determines a leader block's actual status, R_i increases its creator's reputation by 1 for a correct prediction or decreases it by 1 otherwise (Fig. 5, lines 24–28).

This reputation mechanism is intentionally heuristic. Network delays may cause mispredictions even for honest leaders, while Byzantine behavior may produce similar outcomes. Thus, a low score indicates a greater need for recovery checkpoints rather than necessarily identifying a Byzantine replica. Misclassifying an honest replica only incurs additional snapshot overhead without affecting safety.

With this reputation signal, replica R_i records reputation snapshots before speculatively executing potentially risky leader blocks (lines 45–58). Specifically, when a new predicted leader sequence LPs arrives, R_i scans the leaders predicted as to-commit. Before executing each leader B_L, it records a reputation snapshot if B_L's creator ranks among the τ_{rep} lowest-scoring replicas in TR_i (lines 51–52). To handle ties, replicas with identical scores are selected only if the entire tied group falls within the lowest τ_{rep} positions. Each snapshot

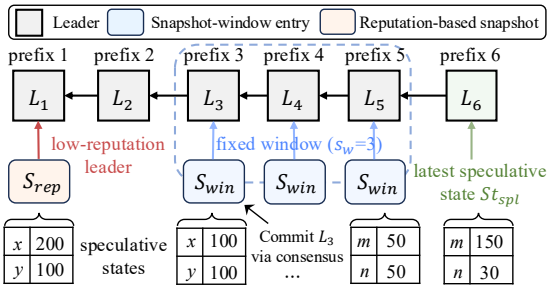


Fig. 4. Commit-aware snapshots: reputation-triggered snapshots $\mathcal{S}_{rep}$ are placed around leader blocks whose creators currently appear less predictable. The snapshot window $\mathcal{S}_{win}$ keeps only the latest s_w speculative prefixes so recently committable states are not lost after later speculative writes.

maps the currently predicted committed-leader prefix to its resulting speculative state. By default, we set $\tau_{rep} = f$, matching the maximum number of Byzantine replicas.

Snapshot window. Pufferfish also retains the most recent s_w speculative prefix snapshots to mitigate overwritten-state issues (lines 56–57). Under continuous speculation, the latest state St_{spl} may include writes from speculative leaders beyond the confirmed prefix, making it unsuitable for directly committing that prefix. For example, in Fig. 4, after speculating through L_6 while consensus has finalized only through L_3, St_{spl} includes writes from L_4–L_6 that may overwrite the state corresponding to L_3. By retaining recent prefix snapshots, the window increases the chance of recovering a state at or before the newly confirmed prefix, allowing direct commitment or shorter replay instead of restarting from the last committed state. In contrast to the straightforward defense, the snapshot window keeps only the newest s_w generic prefix states and discards older ones as speculation moves forward.

This two-component design combines reputation snapshots for higher-risk speculative steps with a bounded window of recent prefix states. However, only the snapshot window has a fixed size: reputation snapshots are pruned as consensus advances but may accumulate during unbounded stalls, and the recovery replay length remains unbounded. Thus, the design offers a practical storage–recovery trade-off rather than fixed worst-case bounds. Simultaneously bounding both storage and recovery costs under unbounded speculation remains an open problem. Our ablation study (§ VI-D) shows that commit-aware snapshotting achieves a decent p99 latency (2.96 s) under adverse stalls, within 16.5% of eager snapshotting (2.54 s) that retain a recovery snapshot after every speculative step, while reducing peak snapshot count by 95.7% and RSS by 50.4% (Table III). In contrast, the latency reaches 28.69 s without snapshots. Commit-aware snapshots therefore preserve most of the recovery benefit of eager snapshotting without retaining the entire speculative history.

D. Transaction Commitment

After consensus finalizes the order, replicas use speculative execution results and snapshots to commit transactions and state transitions. A matching speculative state makes this

Variables:
St_{glb} - The global key-value store state
St_{spl} - The latest speculative state
$\mathcal{S}_{rep}$ - Reputation-triggered snapshots
$\mathcal{S}_{win}$ - Recent speculative-prefix snapshots in the window
s_w - Snapshot-window size for recent speculative prefixes
r_{lc} - The last round where the leader block is committed via consensus
r_{spl} - The round of the last predicted leader block (Fig. 3)
$TR_i[]$ - An array of reputations (indexed by replicas)
$\mathcal{L}_{spl}$ - The APS tree, consisting of a list of LP

23: **upon** receiving an ordered leader block list L_{con} from consensus **do**
24: **for** $LP \in \mathcal{L}_{spl}$ up to $L_{con}[-1].round$ **do** ▷ update reputation
25: **if** $LP.status$ is consistent with the one in L_{con} **then**
26: $TR_i[LP.block.creator]$ += 1
27: **else**
28: $TR_i[LP.block.creator]$ -= 1
29: *consistent* ← true if $\mathcal{L}_{spl}$ is consistent with L_{con} else false
30: **if** *consistent* and $L_{con}[-1].round == r_{spl}$ **then** ▷ Happy path
31: commit_state(St_{glb}, St_{spl})
32: **else**
33: (St_Δ, len) ← find_best_snapshot($L_{con}, \mathcal{S}_{rep}, \mathcal{S}_{win}$)
34: **for** $idx = len$ to $|L_{con}|-1$ **do** ▷ replay suffix beyond snapshot
35: $B_L \leftarrow L_{con}[idx]$
36: txs ← serialize_transactions(B_L) ▷ get ordered txs from leader's causal history
37: St_Δ ←execute(St_Δ, txs, St_{glb})
38: commit_state(St_{glb}, St_Δ)
39: **if not** *consistent* **then** ▷ Fallback path: repair speculative state for future speculation
40: $sequence$ ← the leader predictions in $\mathcal{L}_{spl}$ after $L_{con}[-1]$
41: St_{spl} ← speculative_execution($St_\Delta, sequence$)
42: $r_{lc} \leftarrow L_{con}[-1].round$
43: update_aps_tree(L_{con})
44: update_snapshots($\mathcal{S}_{rep}, \mathcal{S}_{win}, r_{lc}, s_w$)
45: **procedure** speculative_execution(St_{base}, LPs)
46: $St_\Delta \leftarrow St_{base}$ ▷ State transition after executing on LPs
47: $prefix \leftarrow []$
48: **for** LP in LPs **do**
49: **if** $LP.status ==$ to-commit **then**
50: $B_L \leftarrow LP.block$
51: **if** $TR_i[B_L.creator] < TR_i$'s $(f+1)$-th lowest value **then**
52: $\mathcal{S}_{rep}[prefix] \leftarrow St_\Delta$ ▷ snapshot before risky leader
53: txs ← serialize_transactions(LP)
54: St_Δ ←execute(St_Δ, txs, St_{glb})
55: $prefix \leftarrow prefix||[LP]$
56: $\mathcal{S}_{win}[prefix] \leftarrow St_\Delta$ ▷ retain recent speculative prefixes
57: trim_window($\mathcal{S}_{win}, s_w$) ▷ keep only s_w snapshots
58: **return** St_Δ

Fig. 5. Transaction commit for replica R_i in Pufferfish.

post-consensus step lightweight in both normal and delayed operation. Fig. 5 shows the full process for replica R_i. Upon receiving L_{con} from consensus, R_i first compares the actual outcomes with its earlier predictions and updates the reputation table TR_i accordingly (lines 24–28). It then checks whether the speculative order stored in $\mathcal{L}_{spl}$ is consistent with L_{con} (line 29). This consistency test determines whether the replica is in the happy path or the fallback path. After that, the algorithm commits state and resumes speculation.

Happy path. The happy path occurs when the speculative order is correct, i.e., $\mathcal{L}_{spl}$ is consistent with L_{con} (lines 30–38). This is the expected common-case path. In the simplest case, consensus is finalizing exactly the latest speculative prefix, so the latest speculative state St_{spl} can be committed directly (line 31). This is the cheapest outcome because no replay is needed. If the speculative order is correct but the latest speculative state has already advanced beyond the committed prefix, the replica finds the best available snapshot from $\mathcal{S}_{rep} \cup \mathcal{S}_{win}$ and replays only the remaining committed suffix of L_{con} (lines 33–38).

Fallback path. The fallback path occurs only when the speculative order disagrees with L_{con} (lines 32–41). In that case, R_i finds the best available consistent snapshot from $\mathcal{S}_{rep} \cup \mathcal{S}_{win}$ (line 33), starts from that snapshot, and re-executes only the remaining committed suffix of L_{con} before committing. This recovery strategy is deliberately selective. Instead of treating every mis-speculation as a full restart, Pufferfish preserves as much prefix work as possible and repairs only the suffix invalidated by consensus. After that, R_i reconstructs St_{spl} by replaying the remaining predicted suffix on top of the repaired committed base state (line 41).

After commitment, R_i updates the last committed round r_{lc}, prunes decided leaders from the APS tree, and refreshes both snapshot stores (lines 42–44).

Client interface and confirmation. Note that speculative execution is local to each replica, and replicas do not expose speculative results to clients. The client therefore never needs to observe or roll back speculative results. Once consensus fixes a transaction's order and an honest replica has validated and persisted its execution result through either path above, the replica sends the client an authenticated reply identifying the request and its committed result. In Pufferfish, a client accepts the result after receiving $f+1$ matching authenticated replies for that request from distinct replicas. Since there are at most f Byzantine replicas, these replies include at least one from an honest replica, and the client ensures the returned result is correct and committed.

V. Security Analysis

Pufferfish implements BFT SMR: it satisfies the safety and liveness properties of § II-A. Following related work [48], we state them as three properties of the executed state: Serializability, Linearizability, and Liveness. We give sketch proofs here and defer the full proofs to Appendix B. The analysis focuses on the execution layer that Pufferfish adds on top of consensus and builds on two assumptions.

- *(A1) Atomic broadcast.* The dissemination-and-ordering layer is a black-box Byzantine atomic broadcast implemented by the underlying Mysticeti [9]; it delivers to every correct replica the same ordered lists of leader blocks L_{con} (Agreement, Integrity, Validity, and Total Order).
- *(A2) Deterministic and terminating execution.* The function $execute$ (Fig. 5) is deterministic and total: given the same input transaction and the same input state, it always terminates and returns the same output state, as in the EVM [49] and the MoveVM [50].

Throughout, St_0 denotes the genesis state, St_{glb} a replica's committed global state (Fig. 5), and $St^i_{\vdash n}$ the state of replica R_i after its first n committed transactions (§ II-A).

Definition 1 (Pufferfish Serializability)**.** *Let $T = [tx_1, \ldots, tx_n]$ be the sequence of transactions executed by a correct replica. The replica's committed global state satisfies $St_{glb} = \mathrm{seq}(St_0, T)$, where $\mathrm{seq}(\cdot, \cdot)$ denotes a sequential execution from St_0 (Definition 4).*

Theorem 1 (Pufferfish Serializability)**.** *Under (A2), Pufferfish satisfies Definition 1.*

Proof sketch. A central invariant (Lemma 2) shows that the speculative state follows the current predicted leader sequence, while every stored snapshot is the sequential execution of its recorded leader sequence from the last committed state. Prediction correction preserves this invariant by restoring an unchanged prefix and replaying the revised suffix. On each consensus delivery, Fig. 5 therefore commits either the speculative state itself, when the predictions match the delivery exactly, or a snapshot matching a delivery prefix extended by re-executing the remaining committed suffix; in every case, the committed state is the sequential execution of the delivered transaction sequence (Appendix B-B).

Definition 2 (Pufferfish Linearizability)**.** *For any two correct replicas R_i and R_j, there is no n such that both $St^i_{\vdash n}$ and $St^j_{\vdash n}$ are defined and $St^i_{\vdash n} \neq St^j_{\vdash n}$.*

Theorem 2 (Pufferfish Linearizability)**.** *Under (A1)–(A2), Pufferfish satisfies Definition 2.*

Proof sketch. Agreement and Integrity (A1) ensure that every state transition performed at a correct replica is eventually performed exactly once at every correct replica, and Total Order (A1) preserves the relative order of transitions across replicas; with deterministic execution (A2), replicas that have committed the same number of transactions hold identical states (Appendix B-C).

Definition 3 (Pufferfish Liveness)**.** *Every user transaction tx submitted to a correct replica is eventually executed by every correct replica, and its execution result is reflected in St_{glb}.*

Theorem 3 (Pufferfish Liveness)**.** *Under (A1)–(A2), Pufferfish satisfies Definition 3.*

Proof sketch. Validity (A1) guarantees that every transaction submitted to a correct replica is eventually delivered as part of some L_{con}; Fig. 5 terminates on every delivery and persists a committed state that reflects the execution of every delivered transaction (Appendix B-D).

VI. Evaluation

Our evaluation aims to answer the following questions:

- **Performance:** Does overlapping execution with consensus reduce latency across offered loads, replica counts, and crash faults (§ VI-A)?
- **Ordering Stall Resistance:** Can Pufferfish reduce tail latency and recover faster from ordering stalls (§ VI-B)?
- **Byzantine Resilience:** How do Byzantine behaviors affect the performance of Pufferfish (§ VI-C)?
- **Ablation Study:** How do different speculation and snapshot strategies perform under different scenarios (§ VI-D)?

Implementation. We implement Pufferfish in Rust atop the Mysticeti codebase [51]. Since Mysticeti provides only consensus, we integrate Rise EVM [52] for transaction execution, motivated by its well-defined interfaces and the widespread adoption of EVM-compatible transactions. However, Pufferfish is execution-engine agnostic, as its optimizations operate at the system level rather than the instruction level. To isolate the benefits of Pufferfish from parallel execution, we use Rise EVM's sequential execution mode in our evaluation. Our implementation is open source [53].

Baselines. We compare Pufferfish with three baselines: (i) *AutobahnEAC*, which employs Autobahn [15] as its consensus protocol and adopts an execution-after-consensus (EAC) design; (ii) *AutobahnSPC*, which employs Autobahn as its consensus protocol and adopts a speculative-execution design; and (iii) *MysticetiEAC*, which employs Mysticeti [9] as its consensus protocol and adopts an EAC. In the EAC design, transactions are scheduled for execution only after ordering succeeds. In AutobahnSPC's speculative design, replicas speculatively execute transactions after receiving block proposals. All baselines use the same EVM execution engine and transaction scheduler as Pufferfish for a fair comparison.

Datasets. We evaluate Pufferfish using three EVM-compatible smart contract workloads with distinct execution behaviors. *ERC20* consists of simple ERC-20 [54] token transfers. *WETH* uses WETH9 [55] with deposit, approve, transferFrom, and withdraw operations. *Uniswap* executes swaps through a Uniswap-V3-style single-swap contract [56]. Rather than replaying historical Ethereum transactions, we randomly generate transactions to control the offered load and ensure sufficient workload volume. This approach isolates the impact of execution-consensus overlap on confirmation latency across different workloads. All systems use identical generated workloads for fair comparison.

Evaluation conditions. We evaluate three network and ordering conditions: (i) *common*, with normal network operation; (ii) *jitter*, where fluctuating delays may cause isolated or consecutive ordering stalls; and (iii) *adverse*, with consecutive ordering stalls. Although network jitter and consecutive ordering stalls occur naturally in our geo-distributed evaluation environment, their timing depends on transient Internet conditions and cannot be controlled. This variability (i) prevents us from reproducing identical conditions across systems, (ii) impedes reproducibility, and (iii) increases monetary cost by requiring repeated experiments until these uncommon conditions occur. We therefore introduce minimal, controlled perturbations. In the jitter case, we randomly inject message delays into links between replicas. In the adverse case, we induce consecutive ordering stalls by temporarily preventing replicas from performing decision rules while allowing block dissemination to continue.

Experiment setup. To balance evaluation coverage and

cost, we conduct experiments in two environments: a geo-distributed AWS deployment and a local server. The AWS experiments evaluate our primary claims regarding performance (§ VI-A) and ordering stall resistance (§ VI-B), running one time for each evaluation. We use c5a.4xlarge EC2 instances distributed across five AWS regions (us-east-1, us-east-2, us-west-1, eu-west-1, and eu-west-2), with inter-region ping latencies ranging from 50 to 80 ms. Each instance provides 16 vCPUs, 32 GB of RAM, and up to 10 Gbps of bandwidth, running Ubuntu Server 20.04. To reduce experimental costs, we conduct the ablation studies (§ VI-D) and Byzantine fault evaluation (§ VI-C) on a local Ubuntu server equipped with 48 CPU cores and 128 GB of RAM, in which each evaluation runs two times. We deploy one client per replica to submit transactions at a steady offered load and use the default leader timeout specified by each consensus codebase (i.e., 5 seconds in Autobahn and 1 second in Mysticeti). Throughout our evaluation, we set the snapshot window size to 20.

We focus on two performance metrics. *Throughput* is the number of transactions committed per second (TPS). We use *confirmation latency* and *commitment latency* interchangeably to denote the elapsed time from when a client sends a transaction until a replica persists its committed state transition. Unless otherwise specified, all reported metrics are averaged across replicas. As a replica-side metric, this latency excludes reply delivery and the time required for the client to collect $f{+}1$ matching replies (§ IV-D).

A. Performance

We first isolate the common-case benefit of overlapping execution with consensus. We evaluate $n{=}10, 49$ replicas in crash-free and crash-fault configurations using ERC20, varying the offered load until latency rises sharply at saturation.

Crash-free performance. Fig. 6a shows the throughput–latency curves for $n{=}10$ and $n{=}49$. At light load, Pufferfish and MysticetiEAC perform similarly; both sustain 16,000 TPS at approximately 300 ms. At 80,000 TPS with $n{=}10$, MysticetiEAC reaches 681 ms and Pufferfish 430 ms, a 36.9% reduction under normal conditions.

With $n{=}49$ at 25,000 TPS, MysticetiEAC saturates and reaches 724 ms, whereas Pufferfish records 306 ms. Because queuing amplifies this difference, we treat it as a near-saturation result: the pipeline reduces exposure to post-consensus queuing but not the amount of execution work. The Mysticeti-based systems also have lower latency than the Autobahn-based systems, reflecting their shared consensus substrate. At $n{=}49$ and 11,000 TPS, AutobahnEAC reaches 3.76 s, whereas AutobahnSPC remains substantially lower because it moves execution off the post-consensus critical path.

Fig. 6b decomposes latency into consensus latency (block creation to ordering) and non-consensus latency (queuing and execution). At 80,000 TPS, non-consensus latency is substantial for MysticetiEAC. Pufferfish performs the same transaction execution, but performs that work earlier, overlapping it with consensus and reducing post-consensus latency. The benefit is therefore most visible when post-consensus queuing and execution are costly. At light load, these costs are small and the measured benefit is limited.

Performance under crash faults. We next crash one or three of the $n{=}10$ replicas. A crashed leader forces the others to wait for a timeout. As shown in Fig. 6c, the systems have similar latency at lighter loads, with Pufferfish's advantage most pronounced at the highest tested load.

B. Ordering Stall Resistance

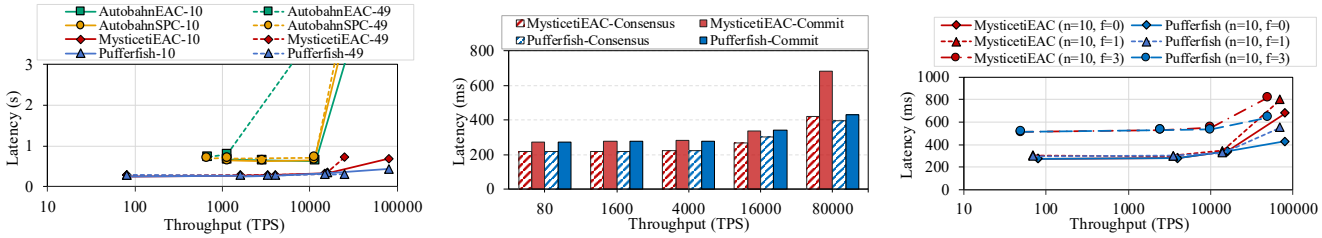
We next evaluate the second benefit of Pufferfish: preserving execution progress as ordering delays extend beyond the common case (Fig. 7). All measurements are performed with a transaction load of 20,000 transactions per second.

Network jitter. We run $n{=}10$ replicas and randomly inject message delays of 0–2 seconds on links between $j{=}3$ replicas. Jitter begins at approximately 10 seconds and ends at approximately 21 seconds. We compute p99 confirmation latency for every one-second interval containing at least one committed transaction. Fig. 7a shows that jitter increases p99 latency and can temporarily prevent both Pufferfish and MysticetiEAC from committing transactions. With $j{=}3$, Pufferfish's first post-jitter commit occurs at $\sim$21 s and MysticetiEAC's at $\sim$24 s. The peak p99 is 10.1 s versus 13.3 s. MysticetiEAC defers execution until ordering succeeds and must then process the accumulated backlog, while Pufferfish can reuse speculative execution results and therefore drains this backlog sooner.

Adverse ordering stalls. We evaluate all four systems under controlled consecutive ordering stalls using the ERC20, WETH, and Uniswap workloads. The ordering stalls continue for approximately 10 s. For each run, we identify the first transaction submitted immediately before the adverse period and measure the time until that transaction is confirmed. Unlike the time-series metric above, this boundary-transaction latency includes the complete interval in which the transaction is blocked and therefore directly reflects recovery speed after consecutive ordering stalls.

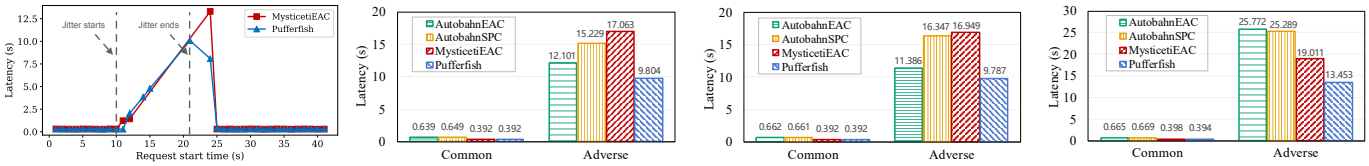
Fig. 7b–7d show that, at 20,000 TPS under common conditions, Pufferfish achieves latency comparable to MysticetiEAC across all workloads. Under adverse conditions, Pufferfish confirms the boundary transaction in 9.80 s, 9.79 s, and 13.45 s for ERC20, WETH, and Uniswap, respectively, reducing confirmation latency by 42.7%, 42.2%, and 29.2% compared to MysticetiEAC. MysticetiEAC's prolonged recovery despite its steady-state throughput of 80,000 TPS (Fig. 6) may reflect CPU contention between backlog execution and consensus catch-up, as well as the cost of persisting a large post-stall committed state transition.

Although all four systems receive the same 20,000 TPS offered load, Autobahn and Mysticeti achieve different dissemination rates under adverse conditions (approximately 11,000 and 16,000 TPS, respectively). Consequently, fewer transactions accumulate behind Autobahn's stalled ordering pipeline, resulting in a smaller execution backlog upon recovery. Thus, while these results reflect end-to-end performance under equal client demand, they do not isolate backlog recovery efficiency.



(a) n=10 and n=49, crash-free (b) n=10, crash-free (c) n=10, crash faults

Fig. 6. Performance evaluation (p99 latency): (a) crash-free performance with different numbers of replicas n; (b) crash-free consensus latency and non-consensus latency with n=10; (c) performance under crash faults.



(a) $n = 10, j = 3$, jitter (b) n =10, ERC20, adverse (c) n =10, WETH, adverse (d) n =10, Uniswap, adverse

Fig. 7. Confirmation latency under evaluation conditions. (a) Jitter condition: missing points denote intervals with no committed transactions. (b–d) Adverse condition: for the ∼10 s adverse stall, we report the latency of the transaction submitted immediately beforehand; lower latency indicates faster recovery.

We therefore focus on the same-consensus comparison between Pufferfish and MysticetiEAC to demonstrate the benefits of speculative execution, while using the Autobahn results to illustrate whole-system behavior. AutobahnSPC further demonstrates that speculation alone does not guarantee faster recovery. Compared to AutobahnEAC, it increases adverse-case latency from 12.1 to 15.23 s for ERC20 and from 11.39 to 16.35 s for WETH, while providing only a marginal improvement for Uniswap (25.77 to 25.29 s). This is because AutobahnSPC speculates on block proposals whose ordering may change during stalls, invalidating speculative work. In contrast, Pufferfish exploits the DAG's ordering stability and commit-aware recovery to preserve speculative work across stalls. The results show that Pufferfish resumes commit progress earlier under jitter and accelerates recovery from ordering stalls.

C. Byzantine Resilience

We finally evaluate how Byzantine behaviors affect the performance of Pufferfish in the local environment. To simulate the realistic network latency, we add a 50-ms delay to message dissemination. We compare Pufferfish with MysticetiEAC under four Byzantine behaviors with Byzantine counts f=1, 3, and under the common and adverse conditions. The following evaluations run with n=10 and a load of 20,000 TPS.

Byzantine behaviors. We consider Byzantine behaviors designed to undermine Pufferfish's effectiveness by misleading initial APS predictions, invalidating speculative execution, or increasing snapshot maintenance overhead. In *leader withholding* (LW), a Byzantine leader withholds its block from all peers. In *selective leader delay* (LD), it delays its block by 2.5 seconds to half of the honest replicas. In *reference withholding* (RW), Byzantine creators omit references to the previous round's honest leader. In *equivocation* (EQ), a Byzantine creator sends two conflicting blocks for the same round to disjoint replicas.

Confirmation latency. We report p99 latency here to assess the impact of Byzantine behaviors on tail latency. (Note that Fig. 7b measure the latency of a transaction exposed to nearly the entire ordering stall to assess recovery speed.) From Fig. 8, we find that the Byzantine behaviors cause latency increases to both Pufferfish and MysticetiEAC, since withholding/delaying messages and equivocation can prolong the decision of leader blocks, impeding the consensus process. However, Pufferfish consistently outperforms MysticetiEAC under Byzantine attacks. Under common conditions, Pufferfish reduces p99 confirmation latency by up to 33.2%. Under adverse conditions, Pufferfish consistently achieves 9.4%–23.4% lower p99 latency across all attacks and fault counts, demonstrating its effectiveness even during consecutive ordering stalls.

Prediction and reuse. Prediction accuracy (hit rate) is the fraction of leader-block predictions matching consensus. Initial/final accuracy implies the first/last prediction before the consensus decision. Table I shows that APS corrects the initial errors induced by selective leader delay and reference withholding, achieving 100% final prediction accuracy in all tested settings even when reference withholding reduces initial accuracy to 90.78%–92.29%. Speculative execution reuse (the percentage of all confirmed sampled transactions observed reusing a matching snapshot) remains nearly 100% under equivocation but drops to 67.27%–78.10% under leader withholding. This drop reflects reduced pre-execution opportunities: APS correctly predicts withheld leaders as to-skip, so no speculative execution is performed for them even though prediction accuracy remains 100%. Nevertheless, Pufferfish maintains competitive p99 latency despite these initial mispredictions (cf. Fig. 8). These experiments demonstrate correction of initial errors; recovery from final mispredictions is not

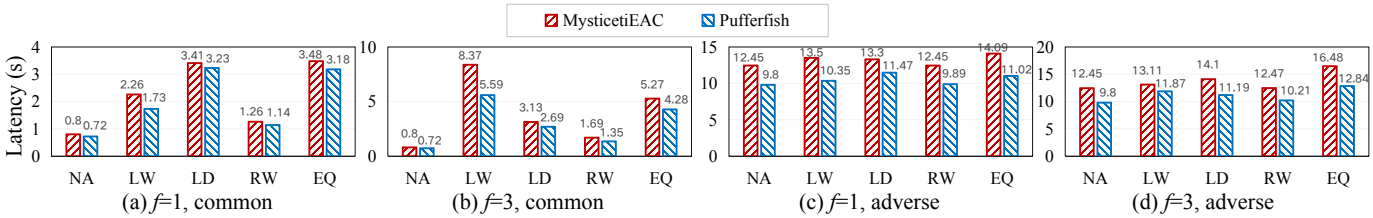


Fig. 8. p99 confirmation latency under distinct Byzantine behaviors with the ERC20 workload. NA represents no attack; the other attacks are defined in § VI-C. Adverse conditions include an imposed 10-second ordering stall.

TABLE I
PUFFERFISH'S MEASUREMENTS UNDER ATTACKS. NA REPRESENTS NO ATTACK; THE OTHER ATTACKS ARE DEFINED IN § VI-C.

Condition	f	APS prediction (%) (initial, final)				Speculative execution reuse (%)					Correction time (ms)				Snapshot counts (window, reputation)			
		LW	LD	RW	EQ	NA	LW	LD	RW	EQ	LW	LD	RW	EQ	LW	LD	RW	EQ
Common	1	(100, 100)	(99.77, 100)	(90.78, 100)	(100, 100)	99.59	72.06	86.42	83.48	99.89	–	348.5	253.5	–	(2, 1)	(5.5, 1)	(8, 2)	(4, 1)
	3	(100, 100)	(99.73, 100)	(91.05, 100)	(100, 100)	99.59	67.27	99.16	84.64	99.90	–	832.5	248	–	(2, 1)	(3.5, 1)	(7, 2)	(4, 1)
Adverse	1	(100, 100)	(100, 100)	(92.29, 100)	(100, 100)	99.37	78.10	92.53	91.64	100.00	–	–	234.5	–	(20, 1)	(20, 6)	(20, 2.5)	(20, 1)
	3	(100, 100)	(99.75, 100)	(91.04, 100)	(100, 100)	99.37	73.36	96.17	88.19	99.58	–	857.5	243.5	–	(20, 3)	(20, 6)	(20, 2)	(20, 1)

exercised in these runs.

Correction and snapshots. Prediction corrections occur within a median of 234.5–857.5 ms across the reported attack settings (Table I). Snapshot counts are averaged across all replicas, hence the fractional values. Under adverse conditions, the peak number of retained snapshots reaches 20 with window-based management, compared to at most 6 with reputation-based management, illustrating the additional state maintained for speculation and recovery.

D. Ablation Study

Having evaluated Pufferfish under normal operation, ordering stalls, and Byzantine behaviors, we now isolate the contributions of its speculation and snapshot policies. In both jitter- and adverse-case runs, the jitter or adverse period lasts for the first 90 seconds. We report p99 confirmation latency because our primary goal is to reduce the latency amplification caused by ordering stalls. Additionally, we report the speculation hit rate and resource consumption.

Speculation strategy. We compare APS with two fixed policies. *All-skip* predicts every undecided leader as to-skip and therefore performs little useful speculative execution when leaders are eventually committed. *All-commit* predicts every undecided leader as to-commit, which exposes more execution opportunities but risks wasted work when leaders are skipped. Our APS instead adapts its prediction to the evidence available in the local DAG. We evaluate n=10 under common conditions and with jitter affecting j=3 and j=5 replicas, and n=30 with j=9 and j=15.

Fig. 9 shows that APS achieves the lowest p99 confirmation latency in every evaluated configuration. For $n = 10$, APS reduces latency from 1.42 to 0.65 s in the common case, from 4.97 to 4.2 s with $j = 3$, and from 12.09 to 8.18 s with $j = 5$, relative to the worse of the two fixed policies in each setting.

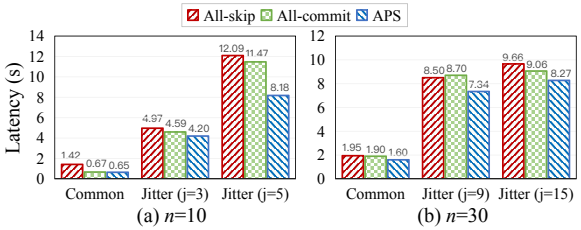


Fig. 9. p99 confirmation latency of different speculation strategies under common-case and network-jitter scenarios.

TABLE II
HIT RATE OF SPECULATION STRATEGIES UNDER JITTERS (%).

	$n = 10$			$n = 30$		
	$j = 0$	$j = 3$	$j = 5$	$j = 0$	$j = 9$	$j = 15$
All-skip	0.5	21.6	27	1.7	13.5	24.2
All-commit	99.9	81	73.3	99.8	83.3	74.4
APS	99.9	92	92.3	99.9	88.1	93.9

These correspond to reductions of 54.2%, 15.5%, and 32.3%, respectively. The same trend holds for $n = 30$.

The initial hit rates (i.e., the fraction of initial leader-block predictions that match the eventual consensus commit/skip decision) in Table II explain these latency reductions. In the common case, all-commit and APS are nearly always correct, whereas all-skip rarely matches the final decision. Under jitter, the accuracy of all-commit falls to 73.3%–83.3% in the more disruptive settings, causing incorrectly speculated work to be discarded. APS maintains a hit rate of 88.1%–93.9% across all jitter settings, improving accuracy by 4.8–19.5 percentage points over the better fixed policy. APS therefore preserves the common-case opportunity to execute early while avoiding much of the wasted execution caused by an unconditional all-commit policy.

TABLE III
EVALUATION OF SNAPSHOT STRATEGIES UNDER ADVERSE CONDITIONS.

	Snapshot count		CPU usage (%)		RSS (MB)	
	com.	adv.	com.	adv.	com.	adv.
EAC	0	0	238.2	172.6	64.8	82.3
None	0	0	241.8	196	68.9	88.6
Eager	4	1885	245.9	201.8	70.9	188.6
Commit-aware	1	81	249.5	204.6	69.7	93.5

Snapshot strategy. We next isolate the snapshot mechanism by comparing four configurations with n=10. *EAC* is the baseline that takes no speculative snapshots. *None* enables speculative execution without maintaining intermediate snapshots, so mis-speculation requires recovery from the committed state. *Eager* snapshots every speculative state. Our policy is *commit-aware*, which selectively retains snapshots that are likely to be useful during recovery.

Table III demonstrates the corresponding resource trade-off under both common and adverse conditions, where snapshot count is the maximum number maintained by a replica, CPU usage is the percentage of one logical CPU core used by a replica, and RSS is resident memory. In the adverse case, eager snapshotting retains up to 1,885 snapshots and consumes 188.6 MB of RSS, compared to 81 snapshots and 93.5 MB for commit-aware snapshotting. This represents 95.7% fewer snapshots and 50.4% lower RSS, with comparable CPU usage. This experiment demonstrates that commit-aware snapshotting achieves a better resource trade-off.

VII. RELATED WORK

Asynchronous BFT protocols. Asynchronous BFT protocols [6], [35]–[37], [57], [58] guarantee probabilistic progress without assuming a known message-delay bound. They therefore avoid timeout-triggered ordering stalls in the partially synchronous sense studied in this paper. However, asynchronous liveness does not imply bounded confirmation latency: an adverse message schedule can still delay ordering or reduce its rate below that of transaction dissemination. If execution waits for a finalized order, such periods can likewise accumulate execution work that must be drained after ordering catches up. Asynchronous consensus and Pufferfish thus address different layers of the problem: the former strengthens ordering-layer progress under uncertain network timing, whereas the latter reduces the execution-layer penalty when a finalized order is delayed. The approaches are complementary, and extending Pufferfish to an asynchronous DAG protocol would primarily require adapting APS to that protocol's commit evidence.

Execute-order-validate SMR schemes. A line of work [28], [38]–[42], [59]–[62] reduces ordering-path overhead by deconstructing the SMR architecture. Execute-order-validate systems, such as Hyperledger Fabric [38] and its successors [39], [40], execute transactions before consensus establishes their global order. Fabric SSI [41] and BIDL [42] further parallelize execution and ordering. In these architectures, pre-execution does not follow a consensus-derived ordering signal. Transactions may therefore be evaluated against state versions that are incompatible with their eventual positions in the finalized order. The validation phase detects such conflicts and invalidates the affected transactions. This order-independent execution allows useful work to proceed while ordering is delayed, but its invalidation rate can increase under contention [63], [64].

Pufferfish takes a different approach. It preserves the ordered-execution semantics of BFT SMR while using DAG evidence to predict the eventual consensus order and execute transactions accordingly. Unlike execute-order-validate, which tolerates conflicts through post-order validation, Pufferfish exploits early ordering signals to improve speculative work reuse. Moreover, execute-order-validate may invalidate already ordered transactions, whereas Pufferfish executes transactions according to the finalized consensus order. These differences in execution semantics make execute-order-validate unsuitable as a drop-in baseline for our setting.

Speculative execution. Speculative execution is widely used to reduce transaction latency [24], [65]–[80]. Zyzzyva [81] lets clients complete after $3f+1$ matching speculative replies or a commit certificate acknowledged by $2f+1$ replicas. Its view-change rules retain histories supported by commit certificates or matching order reports. Proof-of-Execution (PoE) [82], [83] uses $2f+1$ matching execution replies as a proof that the transaction will survive view changes. Both can reuse speculative work retained in a new view, although excluded suffixes may require rollback and normal processing pauses during view installation. Forerunner [77] and its successors [78], [79] pre-execute local transactions before dissemination, but lack global ordering signals and target coupled Ethereum-style consensus. HotStuff-1 [23] and Zaptos [24] instead speculate from block proposals in decoupled BFT; a view change can replace a proposal and invalidate its results, limiting reuse across stalls. MonadBFT [84] can preserve speculative work through tail-forking resistance, although reuse depends on leader responsiveness and network stability.

In Pufferfish, speculative results stay local and clients accept $f+1$ matching replies after replica commitment (§ IV-D). Its execution layer continues extending speculation across consecutive ordering stalls while the DAG grows, reusing matching execution results when consensus resumes.

Pipelined SMR architectures. Many BFT SMR systems [11], [23], [24], [30], [33], [34], [85]–[89] use pipelined architectures that divide transaction processing into stages and overlap them to reduce common-case latency. Pufferfish shares this goal by overlapping consensus and execution, but additionally anchors speculative execution to the evolving DAG and retains prefix states so that pipeline work can remain reusable when ordering stalls invalidate or replace early decisions.

Parallel transaction execution. An orthogonal line of work optimizes transaction execution through concurrent or parallel execution [48], [80], [90]–[97]. This work studies concurrency strategies, transaction-scheduling algorithms, and parallel execution engines that allow a single replica to execute transactions in parallel, thereby reducing execution latency. Because Pufferfish is a system-level optimization, these techniques can be integrated into Pufferfish to further improve transaction

throughput and latency.

VIII. Conclusion

This paper presents Pufferfish to reduce confirmation latency and the execution backlog left by ordering stalls through overlapping consensus with DAG-guided speculative execution. Pufferfish's APS refines order predictions as the DAG grows, while commit-aware snapshots preserve reusable prefix states and reduce replay. Evaluation of our implementation on Mysticeti shows lower common-case latency and faster recovery from consecutive ordering stalls than baselines.

Acknowledgment

The authors would like to thank Dakai Kang for implementing the EVM interface on Mysticeti. This work is partially supported by Mysten Labs and Supra Labs. Jianting Zhang was supported in part by the Bilsland Fellowship.

References

[1] The Sui Team, "The sui blockchain," https://www.sui.io, 2026, accessed: 2026.

[2] The Solana Team, "The solana blockchain," https://solana.com, 2026, accessed: 2026.

[3] The Aptos Team, "The aptos blockchain," https://aptosnetwork.com, 2026, accessed: 2026.

[4] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, 2012, pp. 1–12.

[5] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: a fast and practical deterministic oltp database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2047–2060, 2020.

[6] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *ACM EuroSys*, 2022, pp. 34–50.

[7] S. Duan, H. Zhang, X. Sui, B. Huang, C. Mu, G. Di, and X. Wang, "Dashing and star: Byzantine fault tolerance with weak certificates," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 250–264.

[8] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *ACM CCS*, 2022, pp. 2705–2718.

[9] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian, "Mysticeti: Reaching the limits of latency with uncertified dags," in *Network and Distributed Systems Security Symposium (NDSS)*, 2025.

[10] N. Shrestha, A. Kate, and K. Nayak, "Sailfish: Towards improving latency of dag-based bft," in *IEEE S&P*, 2025.

[11] B. Arun, Z. Li, F. Suri-Payer, S. Das, and A. Spiegelman, "Shoal++: High throughput {DAG}{BFT} can be fast and robust!" in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 813–826.

[12] T. Kichidis, L. Kokoris-Kogias, A. Koshy, I. Sergey, A. Sonnino, M. Tian, and J. Zhang, "Beluga: Block synchronization for bft consensus protocols," *arXiv preprint arXiv:2511.15517*, 2025.

[13] N. Shrestha, Q. Yu, A. Kate, G. Losa, K. Nayak, and X. Wang, "Optimistic, signature-free reliable broadcast and its applications," *arXiv preprint arXiv:2505.02761*, 2025.

[14] A. Spiegelman, B. Aurn, R. Gelashvili, and Z. Li, "Shoal: Improving dag-bft latency and robustness," in *International Conference on Financial Cryptography and Data Security*. Springer, 2024, p. 92–109.

[15] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks, "Autobahn: Seamless high speed bft," *ACM SOSP*, 2024.

[16] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire." in *NSDI*, vol. 8, 2008, pp. 189–204.

[17] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," in *25th International Conference on Principles of Distributed Systems*, 2022.

[18] T. Crain, C. Natoli, and V. Gramoli, "Red belly: A secure, fair and scalable open blockchain," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 466–483.

[19] F. C. Geyer, H.-A. Jacobsen, R. Mayer, and P. Mandl, "An end-to-end performance comparison of seven permissioned blockchain systems," in *Proceedings of the 24th international middleware conference*, 2023, pp. 71–84.

[20] M. J. Amiri, C. Wu, D. Agrawal, A. El Abbadi, B. T. Loo, and M. Sadoghi, "The bedrock of byzantine fault tolerance: A unified platform for {BFT} protocols analysis, implementation, and experimentation," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 371–400.

[21] V. Gramoli, R. Guerraoui, A. Lebedev, and G. Voron, "Stabl: The sensitivity of blockchains to failures," in *Proceedings of the 26th International Middleware Conference*, 2025, pp. 202–214.

[22] A. Lebedev and V. Gramoli, "Blockchain communication vulnerabilities," *arXiv preprint arXiv:2603.02661*, 2026.

[23] D. Kang, S. Gupta, D. Malkhi, and M. Sadoghi, "Hotstuff-1: Linear consensus with one-phase speculation," *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, pp. 1–29, 2025.

[24] Z. Xiang, Z. Li, B. Arun, T. Zhang, and A. Spiegelman, "Zaptos: Towards optimal blockchain latency," *arXiv preprint arXiv:2501.10612*, 2025.

[25] G. Tsimos, A. Kichidis, A. Sonnino, and L. Kokoris-Kogias, "Hammerhead: Leader reputation for dynamic scheduling," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2024, pp. 1377–1387.

[26] S. Cohen, R. Gelashvili, L. Kokoris-Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, "Be aware of your leaders," in *International Conference on Financial Cryptography and Data Security*. Springer, 2022.

[27] S. Cohen, G. Goren, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Proof of availability and retrieval in a modular blockchain architecture," in *International Conference on Financial Cryptography and Data Security*. Springer, 2023, pp. 36–53.

[28] J. Zhang, Z. Luo, R. Ramesh, and A. Kate, "Optimal sharding for scalable blockchains with deconstructed smr," *Proceedings of the VLDB Endowment*, 2025.

[29] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, apr 1988.

[30] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM symposium on principles of distributed computing*, 2019, pp. 347–356.

[31] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *FC*. Springer, 2022, pp. 296–315.

[32] A. Tonkikh, B. Arun, Z. Xiang, Z. Li, and A. Spiegelman, "Raptr: Prefix consensus for robust high-performance bft," *arXiv preprint arXiv:2504.18649*, 2025.

[33] D. Malkhi and K. Nayak, "Hotstuff-2: Optimal two-phase responsive bft," *Cryptology ePrint Archive*, 2023.

[34] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, "Fast-hotstuff: A fast and resilient hotstuff protocol," *arXiv preprint arXiv:2010.11454*, 2020.

[35] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *ACM CCS*, 2016, pp. 31–42.

[36] P. Jovanovic, L. K. Kogias, B. Kumara, A. Sonnino, P. Tennage, and I. Zablotchi, "Mahi-mahi: Low-latency asynchronous bft dag-based consensus," *arXiv preprint arXiv:2410.08670*, 2024.

[37] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1187–1201.

[38] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[39] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 105–122.

[40] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in

Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 543–557.

[41] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran, "Blockchain meets database: design and implementation of a blockchain relational database," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1539–1552, 2019.

[42] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au *et al.*, "Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks," in *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*, 2021, pp. 18–34.

[43] Sui Engineering Team, "Private communication on the June 2025 sui testnet incident," 2025, private communication with the Sui engineering team, June 2025.

[44] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino *et al.*, "Sui lutris: A blockchain combining broadcast and consensus," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2606–2620.

[45] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 17–33.

[46] J. Zhang and A. Kate, "No fish is too big for flash boys! frontrunning on dag-based blockchains," in *2025 IEEE Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2025, pp. 1065–1080.

[47] J. Zhang, S. Yang, A. Sonnino, S. Loza, and A. Kate, "Lifefin: Escaping mempool explosions in dag-based bft," *arXiv preprint arXiv:2511.15936*, 2025.

[48] Q. Kniep, L. Kokoris-Kogias, A. Sonnino, I. Zablotchi, and N. Zhang, "Pilotfish: Distributed execution for scalable blockchains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2025, pp. 287–306.

[49] E. Fundation, "Ethereum virtual machine (evm)," https://ethereum.org/en/developers/docs/evm/, accessed: 2026.

[50] Sui, "Sui virtual machine," https://docs.sui.io/concepts/sui-move-concepts, accessed: 2026.

[51] Alberto Sonnino, "Mysticeti codebase," github.com/asonnino/mysticeti, 2026.

[52] RiseLabs, "Rise parallel evm," https://github.com/risechain/pevm, 2026.

[53] Jianting Zhang, "Pufferfish codebase," https://github.com/EtherCS/SpeculativeDAG/, 2026.

[54] F. Vogelsteller and V. Buterin, "ERC-20: Token standard," Ethereum Improvement Proposals, EIP-20, 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[55] Ethereum, "Wrapped ether (weth)," https://ethereum.org/wrapped-eth/, accessed: 2026.

[56] U. team, "Uniswap v3 Core," 2021. [Online]. Available: https://app.uniswap.org/whitepaper-v3.pdf

[57] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All You Need is DAG," in *PODC'21: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021.

[58] R. Huang, X. Wang, X. Yan, L. Fan, G. Xue, and S. Liu, "Chitu: Avoiding unnecessary fallback in byzantine consensus," in *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, 2025, pp. 923–942.

[59] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 253–267.

[60] Z. Peng, Y. Zhang, Q. Xu, H. Liu, Y. Gao, X. Li, and G. Yu, "Neuchain: a fast permissioned blockchain system with deterministic ordering," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2585–2598, 2022.

[61] S. Guan, R. Zhang, Q. Ding, M. Song, Z. Xiao, J. Long, M. Wan, T. Yuan, and J. Dong, "Nexus: A novel transaction processing framework for permissioned blockchain," *IEEE Transactions on Parallel and Distributed Systems*, 2026.

[62] N. Shrestha and A. Kate, "Towards improving throughput and scalability of dag-based bft smr," in *ACM EuroSys*, 2026.

[63] Z. Chen, H. Zhuo, Q. Xu, X. Qi, C. Zhu, Z. Zhang, C. Jin, A. Zhou, Y. Yan, and H. Zhang, "Schain: A scalable consortium blockchain exploiting intra-and inter-block concurrency," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 2799–2802, 2021.

[64] J. Wen, W. Chen, J. Zhang, X. Chen, T. Cai, H.-N. Dai, and Z. Zheng, "Rapidsnail: Improve scalability of blockchain under high contention workload," *IEEE Transactions on Computers*, 2025.

[65] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," *ACM SIGOPS operating systems review*, vol. 39, no. 5, pp. 191–205, 2005.

[66] N. Neelakantam, D. R. Ditzel, and C. Zilles, "A real system evaluation of hardware atomicity for software speculation," *ACM Sigplan Notices*, vol. 45, no. 3, pp. 29–38, 2010.

[67] C. Tian, M. Feng, and R. Gupta, "Supporting speculative parallelization in the presence of dynamic data structures," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 62–73.

[68] J. W. Mickens, J. R. Howell, J. R. Lorch, J. E. Elson, and E. B. Nightingale, "Network application performance enhancement using speculative execution," Mar. 20 2012, uS Patent 8,140,646.

[69] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve:{Execute-Verify} replication for {Multi-Core} servers," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 237–250.

[70] X. Yan, A. P. Joa, B. Wong, B. Cassell, T. Szepesi, M. Naouach, and D. Lam, "Specrpc: A general framework for performing speculative remote procedure calls," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 266–278.

[71] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D.-J. Shin, T. Kim, and B.-G. Chun, "Speculative symbolic graph execution of imperative deep learning programs," *ACM SIGOPS Operating Systems Review*, vol. 53, no. 1, pp. 26–33, 2019.

[72] L. Liu, T. Millstein, and M. Musuvathi, "Accelerating sequential consistency for java with speculative compilation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 16–30.

[73] J. Qiu, L. Jiang, and Z. Zhao, "Challenging sequential bitstream processing via principled bitwise speculation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 607–621.

[74] J. Behrens, A. Cao, C. Skeggs, A. Belay, M. F. Kaashoek, and N. Zeldovich, "Efficiently mitigating transient execution attacks using the unmapped speculation contract," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1139–1154.

[75] Y. Xia, P. Jiang, and G. Agrawal, "Scaling out speculative execution of finite-state machines with parallel merge," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 160–172.

[76] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 707–720.

[77] Y. Chen, Z. Guo, R. Li, S. Chen, L. Zhou, Y. Zhou, and X. Zhang, "Forerunner: Constraint-based speculative transaction execution for ethereum," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 570–587.

[78] R. Pan, C. Liu, G. Xiao, M. Duan, K. Li, and K. Li, "An algorithm and architecture co-design for accelerating smart contracts in blockchain," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[79] S. Zhang, R. Cheng, X. Liu, J. Xiao, H. Jin, and B. Li, "Seer: Accelerating blockchain transaction execution by fine-grained branch prediction," *Proceedings of the VLDB Endowment*, vol. 18, no. 3, pp. 822–835, 2024.

[80] T. Xu, Y. Zhong, Y. Zhang, R. Xiong, J. Zhang, G. Xue, and S. Liu, "Vegeta: Enabling parallel smart contract execution in leaderless blockchains," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 795–811.

[81] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.

[82] J. Hellings, S. Gupta, S. Rahnama, J. Chen, C. Sana, and M. Sadoghi, "Proof-of-execution: Low-latency consensus via speculative execution," *ACM Transactions on Database Systems*, vol. 51, no. 3, pp. 18:1–18:48, 2026.

[83] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi, "Proof-of-execution: Reaching consensus through fault-tolerant speculation," *arXiv preprint arXiv:1911.00838*, 2019.
[84] M. M. Jalalzai, K. Babel, J. Komatovic, T. Klenze, S. Das, F. Elsheimy, M. Setrin, J. Bergschneider, and B. Gilkalaye, "Monadbft: Fast, responsive, fork-resistant streamlined consensus," *arXiv preprint arXiv:2502.20692*, 2025.
[85] B. Y. Chan and E. Shi, "Streamlet: Textbook streamlined blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 1–11.
[86] S. Satija, A. Mehra, S. Singanamalla, K. Grover, M. Sivathanu, N. Chandran, D. Gupta, and S. Lokam, "Blockene: A high-throughput blockchain over mobile devices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 567–582.
[87] C. Xu, C. Zhang, J. Xu, and J. Pei, "Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2314–2326, 2021.
[88] I. Doidge, R. Ramesh, N. Shrestha, and J. Tobkin, "Moonshot: Optimizing block period and commit latency in chain-based rotating leader bft," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 470–482.
[89] W. Chen, D. Xia, Z. Cai, H.-N. Dai, J. Zhang, Z. Hong, J. Liang, and Z. Zheng, "Porygon: Scaling blockchain via 3d parallelism," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 1944–1957.
[90] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 232–244.
[91] Y. Hay and R. Friedman, "Batch-schedule-execute: on optimizing concurrent deterministic scheduling for blockchains," in *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2024, pp. 163–174.
[92] Z. Chen, T. Yang, Y. Zheng, Z. Zhang, C. Jin, and A. Zhou, "Spectrum: Speedy and strictly-deterministic smart contract transactions for blockchain ledgers," *Proceedings of the VLDB Endowment*, vol. 17, no. 10, pp. 2541–2554, 2024.
[93] D. Ryu and C. Park, "Toward high-performance blockchain system by blurring the line between ordering and execution," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1–16.
[94] J. Chen, A. Sonnino, L. Kokoris-Kogias, and M. Sadoghi, "Thunderbolt: Concurrent smart contract execution with non-blocking reconfiguration for sharded dags," *arXiv preprint arXiv:2407.09409*, 2024.
[95] H. Lin, H. Feng, Y. Zhou, and L. Wu, "Parallelevm: Operation-level concurrent transaction execution for evm-compatible blockchains," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 211–225.
[96] H. Wang, M. Pan, and J. Wang, "Crystality: a programming model for smart contracts on parallel evms," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025, pp. 412–425.
[97] P. S. Anjana, M. Amini, R. Kapoor, R. Parmar, R. Ramesh, S. Ravi, and J. Tobkin, "Efficient parallel execution of blockchain transactions leveraging conflict specifications," in *7th Conference on Advances in Financial Technologies (AFT 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025, pp. 29–1.
[98] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

APPENDIX A
DECOUPLED BFT CONSENSUS UNDER ORDERING STALLS

In this section, we compare representative decoupled BFT designs and explain why we choose Mysticeti as the substrate for Pufferfish. The comparison characterizes the cited protocols rather than claiming that every protocol in a family has the same behavior. For common-case latency, a useful substrate must expose ordering evidence early enough to overlap execution with consensus; under stalls, that evidence should remain useful as ordering evolves across rounds. Fig. 10 illustrates these properties in three representative designs.

Batch-based BFT protocols (Fig. 10(a)) disseminate transaction batches independently of ordering, either directly [23], [30], [31], [33] or with availability certificates [24], [28], [32], [62]. A chain-based ordering protocol then proposes an ordered list of batch digests. Such a proposal provides an early signal for common-case speculation, as exploited by proposal-guided designs such as HotStuff-1 [23] and Zaptos [24].

The reuse challenge arises when a view change replaces an uncommitted proposal. In Fig. 10(a), OP_3 orders bt_4 after bt_3, but its ordering attempt stalls; the replacement OP_4 selects bt_5 instead. Speculation tied to OP_3 may therefore be invalidated. Thus, proposal-guided execution can shorten the common-case path, but the representative replacement rule does not itself preserve speculative order across consecutive stalls. A different batch-based protocol could expose an additional continuity mechanism.

Multichain BFT protocols (Fig. 10(b)) let each replica extend its own available chain in parallel [15]. An ordering proposal contains a *chain cut*, comprising the latest selected block from each chain; a deterministic rule linearizes the uncommitted prefixes represented by that cut. For example, OP_3 uses $cut_2 = (A_4, B_4, C_3, D_4)$ in Fig. 10(b).

As with batch proposals, a replacement chain cut can imply a different order. After OP_3 stalls, OP_4 proposes cut_3 over newer chain tips, changing the order associated with the abandoned proposal. This is the reuse limitation tested by AutobahnSPC in § VI-B: proposal-guided speculation alone provides no consistent recovery benefit over AutobahnEAC across our three workloads. This experiment does not establish an impossibility for all multichain protocols; it shows that effective cross-stall speculation requires either continuity across cuts or another explicit reuse mechanism.

DAG-based BFT protocols (Fig. 10(c)) disseminate blocks in logical rounds, with each block referencing a quorum of blocks from the preceding round. The resulting append-only DAG exposes causal and voting evidence before leaders are finally decided, giving APS an early signal from which to derive a speculative order (§ IV-A).

The same causal structure can preserve speculative prefixes as ordering evolves. Fig. 10(c) illustrates this behavior in Mysticeti [9] using leader blocks B_2 and A_5. Although B_2 is not decided directly, the later committed leader A_5 includes evidence for B_2 in its causal history and decides it indirectly. Because A_5 extends the order implied by B_2, the speculative prefix derived from B_2 remains reusable rather than being replaced.

The distinction is therefore not dissemination decoupling alone: all three designs can disseminate data independently of final ordering. Proposal-based evidence can enable common-case speculation, whereas cross-stall reuse additionally depends on whether later ordering evidence preserves the predicted prefix. DAG protocols provide this continuity through their growing causal histories, making the Pufferfish architecture applicable beyond Mysticeti. The protocol-specific component is APS: each DAG protocol has different leader schedules, voting patterns, and commit rules, so its predictor must translate the corresponding local evidence into likely ordering outcomes.

APPENDIX B
SECURITY PROOFS

This appendix proves the three properties stated in § V, Serializability, Linearizability[1], and Liveness, which together ensure the safety and liveness statements of § II-A. We first state the assumptions and formal machinery, then prove each property in turn.

A. Assumptions and Notation

Pufferfish guarantees properties of the executed *state*, implementing BFT SMR. We assume all replicas start from a fixed *genesis* state St_0, and let St_{glb} denote a replica's committed global state after Fig. 5 has processed all consensus deliveries received so far. We write $St^i_{\vdash n}$ for the global state of replica R_i at the moment its cumulative committed-transaction count first reaches n; this is well-defined precisely when n is a delivery-boundary count at R_i (i.e., the cumulative count after some $commit_state$ invocation), since St_{glb} is updated atomically per delivery.

Atomic broadcast. Following related work [9], [44], [48], we treat the dissemination-and-ordering layer as a black-box *Byzantine atomic broadcast* (BAB) primitive that the underlying BFT consensus [9] implements. A correct replica R_k broadcasts a message m by calling $a_broadcast_k(m)$, and each correct replica R_i outputs $a_deliver_i(m)$ when m is delivered. The primitive satisfies reliable broadcast (Agreement, Integrity, Validity) and a total-order property [9]:

- *Agreement:* If a correct replica R_i outputs $a_deliver_i(m)$, every correct replica R_j eventually outputs $a_deliver_j(m)$.
- *Integrity:* A correct replica R_i outputs $a_deliver_i(m)$ at most once for each message m.
- *Validity:* If a correct replica R_k calls $a_broadcast_k(m)$, every correct replica R_i eventually outputs $a_deliver_i(m)$.

[1]Classical linearizability [98] is defined per object or unit of state: for every unit of state, the sequence of operations affecting it appears identical across executions. Object-store systems such as Pilotfish [48] adopt this formulation directly, since their state is structured as a set of objects. Pufferfish instead reasons about an abstract global state St_{glb} that subsumes the object decomposition: when St_{glb} is instantiated as a collection of objects, Definition 2 implies the per-object formulation, since agreement on the global state entails agreement on the per-object sub-states it contains.

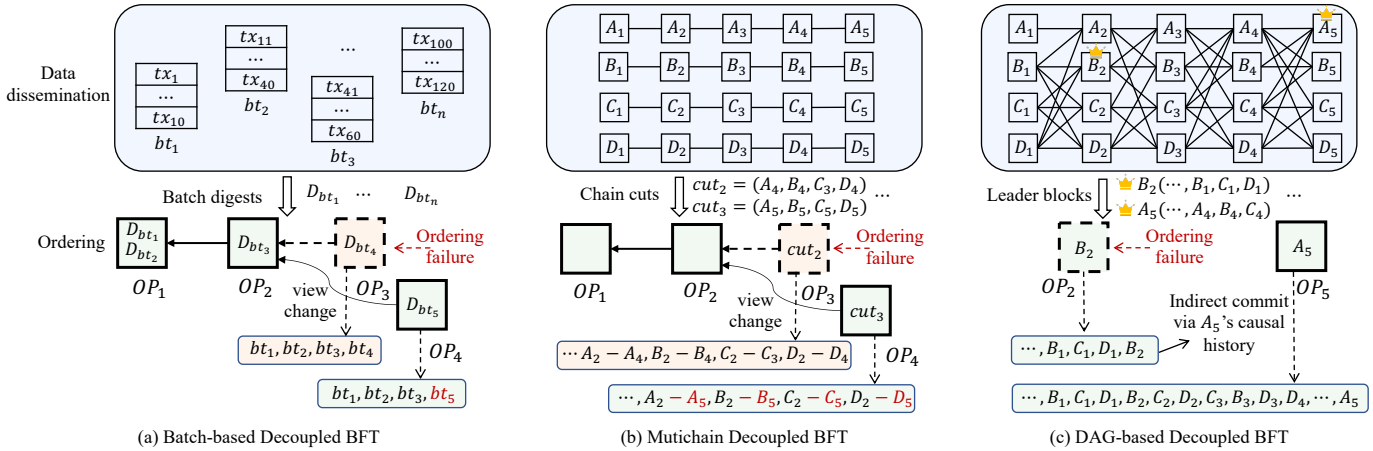


Fig. 10. Comparison of representative decoupled BFT consensus designs under ordering stalls. The top shows data dissemination; the bottom shows ordering proposals OP and their implied orders. Replacement proposals can change the speculative order in (a) and (b), whereas the later DAG leader in (c) preserves the earlier causal prefix.

- *Total order:* If a correct replica R_i outputs $a_deliver_i(m)$ before $a_deliver_i(m')$, then no correct replica R_j outputs $a_deliver_j(m')$ before $a_deliver_j(m)$.

In Pufferfish, BAB messages are *consensus deliveries* L_{con}: each $a_deliver$ event at a correct replica yields an ordered list of leader blocks of transactions, processed by Fig. 5.

Assumptions. The proofs build on two assumptions:

- *(A1) Atomic broadcast.* The underlying Mysticeti consensus protocol [9] provides Byzantine atomic broadcast as defined above.
- *(A2) Deterministic and terminating execution.* The function $execute$ used by Pufferfish (Fig. 5) is deterministic and total: given the same input transaction tx and the same input state, it always terminates and returns the same output state. This is satisfied by standard execution engines such as the EVM [49] and the MoveVM [50], where termination is enforced by gas metering.

Modeling conventions. Throughout the proofs, we treat every state (St_{glb}, St_{spl}, St_{Δ}, and the snapshot values) as a complete state, and we model $execute$ as a two-argument function of an input state and a transaction sequence. The third argument St_{glb} passed to $execute$ in Fig. 5 is a read-through implementation optimization—it lets the execution engine resolve reads of data not materialized in the speculative delta—and has no semantic effect on the output state. Furthermore, the three event handlers of Fig. 3 and Fig. 5 execute atomically and sequentially at each correct replica; in particular, no speculative-execution step interleaves with the processing of a consensus delivery.

B. Proof of Serializability

We first formalize what it means to execute a sequence of transactions sequentially, then establish an invariant relating the committed state, the speculative state, and the snapshot stores (Lemma 2), and finally derive Theorem 1 by induction over consensus deliveries.

Definition 4 (Sequential schedule)**.** *A sequential schedule applies 'execute' to a sequence of transactions $T = [tx_1, \ldots, tx_n]$ in order, where the output state of tx_i is the input state of tx_{i+1}, starting from a given state St. We denote the resulting state by* $\mathrm{seq}(St, T)$*, with* $\mathrm{seq}(St, []) = St$ *and the associativity property* $\mathrm{seq}(St, T_1 \,\|\, T_2) = \mathrm{seq}(\mathrm{seq}(St, T_1), T_2)$ *for any sequences T_1, T_2, where $\|$ denotes sequence concatenation.*

For a list of leader predictions LPs, let $T(LPs)$ denote the concatenation of $serialize_transactions(LP)$ over the to-commit entries of LPs, in the order of LPs. For any block sequence K, including a delivery or a snapshot key, we write $T(K)$ for the analogous flattened transaction sequence, and $L_{con}[1..\ell]$, with $0 \le \ell \le |L_{con}|$, for the list of its first ℓ leader blocks. The linearization of a leader block's causal history is deterministic and depends only on the sequence of previously ordered leader blocks (§ IV-A); consequently, two lists carrying the same blocks in the same order induce the same flattened transaction sequence in the same committed context, whether they arise from predictions or from a delivery.

Recall from Fig. 5 that r_{lc} denotes the replica's last committed leader round (initially the genesis round). We call the leader predictions of $\mathcal{L}_{spl}$ for rounds greater than r_{lc} the *pending predictions*, denoted P: the pruning of decided leaders (Fig. 5, line 43) removes predictions covered by a delivery, so P is exactly the speculation performed since the last commit. A snapshot key records the predicted to-commit block sequence at its creation, relative to the last commit.[2]

Specification of the snapshot helpers. The proofs use the following contracts for the helper procedures of

[2]In Fig. 5, the variable $prefix$ is re-initialized per invocation of $speculative_execution$ for readability; for chained invocations, the key is the accumulated prefix of the chain. Commit-handler repairs prefix new keys with L_{con} until cleanup.

Figs. 3 and 5 whose pseudocode is not shown. The call $find_best_snapshot$ (line 33) returns $(St_K, |K|)$ for some entry (K, St_K) of $\mathcal{S}_{rep} \cup \mathcal{S}_{win}$ whose key equals a prefix of L_{con} as a block sequence (preferring the longest such entry), and returns $(St_{glb}, 0)$, i.e., the last committed state, if no stored key matches. At commit time, $update_snapshots$ (line 44) retains exactly the entries whose keys extend the block sequence of the freshly committed L_{con}, re-keying each retained entry $L_{con} \,\|\, K'$ to K'; $trim_window$ only removes entries. Neither procedure modifies stored states.

The helper $refresh_speculation_from(r, \mathcal{L}_{spl})$ (line 17) restores a snapshot (K, St_K) whose key matches a prefix of the predicted to-commit blocks before round r, or $([], St_{glb})$ if none matches. It replays the revised predictions after K through $speculative_execution$, restoring the prefix's linearization context and initializing snapshot keys with K. It returns the resulting state without modifying St_{glb}.

We next make the consistency test of Fig. 5 (line 29) precise.

Definition 5 (Consistency)**.** *The APS tree $\mathcal{L}_{spl}$ is* consistent *with a delivered L_{con} if and only if (i) $\mathcal{L}_{spl}$ contains a leader prediction for every leader round r with $r_{lc} < r \leq L_{con}[-1].round$, and (ii) the blocks of those predictions with status to-commit are exactly the leader blocks of L_{con}, in delivery order. In particular, if a delivery extends beyond the last predicted round (i.e., $L_{con}[-1].round > r_{spl}$), some prediction required by (i) is missing and the test fails; the replica then takes the snapshot-and-replay branch, which is always safe.*

The following lemma captures the invariant of speculative execution (Fig. 5, § IV-C).

Lemma 1 (Speculative-execution invariant)**.** *Under (A2), invoking $speculative_execution$ on any base state St and any list of leader predictions LPs returns* $\mathrm{seq}(St, T(LPs))$.

Proof. Induction on $|LPs|$. For $|LPs| = 0$, the loop at Fig. 5, lines 48–57 is empty and the procedure returns $St_\Delta = St = \mathrm{seq}(St, [])$. For the inductive step, let $LPs = LPs' \,\|\, [LP]$ and assume the lemma holds for LPs'. After processing LPs', the loop variable St_Δ equals $\mathrm{seq}(St, T(LPs'))$. There are two cases for LP. (a) If $LP.status$ = to-skip, the inner branch is not taken, $T([LP]) = []$, and St_Δ is unchanged. (b) If $LP.status$ = to-commit, Fig. 5, lines 53–54, computes $\mathrm{txs} = serialize_transactions(LP)$ and $St_\Delta \leftarrow execute(St_\Delta, \mathrm{txs}, \cdot)$. By (A2) and Definition 4, this assigns

$$\begin{aligned} St_\Delta &= \mathrm{seq}(\mathrm{seq}(St, T(LPs')), \mathrm{txs}) \\ &= \mathrm{seq}(St, T(LPs') \,\|\, \mathrm{txs}) = \mathrm{seq}(St, T(LPs)) \end{aligned}$$

by associativity. The procedure then returns St_Δ. □

We now establish the central invariant relating the committed state, the speculative state, and the snapshot stores.

Lemma 2 (Speculation invariants)**.** *Under (A2), the following invariants hold at every correct replica, where St_{glb} denotes its committed state and P its pending predictions:*

- *(a) At every point between handler executions, the speculative state satisfies $St_{spl} = \mathrm{seq}(St_{glb}, T(P))$.*
- *(b) At every point between handler executions, every snapshot entry $(K, St_K) \in \mathcal{S}_{rep} \cup \mathcal{S}_{win}$ satisfies $St_K = \mathrm{seq}(St_{glb}, T(K))$.*
- *(c) Processing a delivery L_{con} from committed state St_{glb}^{prev} sets the committed state to $\mathrm{seq}(St_{glb}^{prev}, T(L_{con}))$.*

Proof. By induction over the sequence of (atomic, sequential) handler executions at the replica. Initially $St_{spl} = St_{glb} = St_0$, $P = []$, and both snapshot stores are empty, so (a) holds since $\mathrm{seq}(St_0, []) = St_0$ and (b) holds vacuously.

Speculation handler (Fig. 3). The handler appends the newly predicted $sequence$ to $\mathcal{L}_{spl}$, extending the pending predictions to $P \,\|\, sequence$, and advances St_{spl} via speculative execution (Fig. 3, line 11). By the induction hypothesis (a) and Lemma 1, the new speculative state is

$$\begin{aligned} &\mathrm{seq}(\mathrm{seq}(St_{glb}, T(P)), T(sequence)) \\ &\qquad = \mathrm{seq}(St_{glb}, T(P \,\|\, sequence)) \end{aligned}$$

by associativity, so (a) is preserved. For (b), pre-existing entries are untouched. A new entry inserted while processing an element LP of $sequence$ has as key the accumulated to-commit prefix K strictly before LP (for a reputation snapshot, inserted before the $execute$ call, lines 51–52) or including LP (for a window snapshot, inserted after it, lines 56–57), and stores the loop state St_Δ at the moment of insertion, which equals $\mathrm{seq}(St_{glb}, T(K))$ by the induction hypothesis (a), Lemma 1 applied to the prefix of $sequence$ processed so far, and associativity. Hence (b) is preserved.

Correction handler (Fig. 3, lines 12–17). If the guard is false, nothing changes. Otherwise, let P' be the revised predictions, K the restored prefix, and U the remaining predictions after K. Since K precedes the corrected round, it is an unchanged prefix of the to-commit blocks of P'. By (b), $St_K = \mathrm{seq}(St_{glb}, T(K))$ (also for the default empty key). Thus Lemma 1 and associativity give

$$St_{spl} = \mathrm{seq}(St_K, T(U)) = \mathrm{seq}(St_{glb}, T(P')),$$

establishing (a). Retained snapshots still satisfy (b) because the committed base is unchanged; snapshots created during replay satisfy it by Lemma 1 with accumulated prefix keys. The committed state is unchanged, so (c) is unaffected.

Commit handler (Fig. 5), for a delivery L_{con} with $T_{new} = T(L_{con})$. Let St_{glb}^{prev} and P^{prev} denote the values at the start of the handler; the reputation update (lines 24–28) touches neither states nor snapshot stores. We prove (c) by distinguishing three cases along the consistency test and the happy-path guard.

Case 1: consistent and $L_{con}[-1].round = r_{spl}$ (direct commit, line 31). Since Fig. 3 predicts every leader round in order, P^{prev} covers exactly the rounds in $(r_{lc}, r_{spl}]$, and the guard equates r_{spl} with the delivery frontier $L_{con}[-1].round$. By Definition 5, the to-commit blocks of P^{prev} are then exactly the leader blocks of L_{con}, in order, so $T(P^{prev}) = T_{new}$. By the induction hypothesis (a), $commit_state$ persists $St_{spl} =$

$\text{seq}(St_{glb}^{prev}, T(P^{prev})) = \text{seq}(St_{glb}^{prev}, T_{new})$, establishing (c).

Case 2: consistent and $L_{con}[-1].round \neq r_{spl}$, i.e., speculation has advanced beyond the delivery ($r_{spl} > L_{con}[-1].round$; recall from Definition 5 that a delivery beyond r_{spl} is inconsistent), and *Case 3: inconsistent* (both use snapshot and replay, lines 33–38). By the specification of $find_best_snapshot$ and the induction hypothesis (b), the returned pair (St_Δ, ℓ) satisfies $St_\Delta = \text{seq}(St_{glb}^{prev}, T(L_{con}[1..\ell]))$: either St_Δ is a stored value whose key equals $L_{con}[1..\ell]$ as a block sequence, or it is the default $(St_{glb}^{prev}, 0)$ and $\text{seq}(St_{glb}^{prev}, []) = St_{glb}^{prev}$. The replay loop (lines 34–37) then applies $execute$ to the remaining leader blocks of L_{con} in order, so by (A2), Definition 4, and associativity, $St_\Delta = \text{seq}(St_{glb}^{prev}, T_{new})$ upon loop exit, and $commit_state$ persists it (line 38), establishing (c).

It remains to show that (a) and (b) are re-established at the end of the handler, where $St_{glb} = \text{seq}(St_{glb}^{prev}, T_{new})$ by (c), and the pruning at line 43 makes the new pending predictions P the suffix of P^{prev} for rounds beyond $L_{con}[-1].round$. For (a): in Case 1, $P = []$ and $St_{spl} = St_{glb}$. In Case 2, St_{spl} is unchanged and, by the induction hypothesis (a) and consistency, $St_{spl} = \text{seq}(St_{glb}^{prev}, T(P^{prev}))$ equals $\text{seq}(St_{glb}^{prev}, T_{new} \,\|\, T(P))$, hence $\text{seq}(St_{glb}, T(P))$ by associativity. In Case 3, line 41 recomputes St_{spl} by speculative execution from $St_\Delta = St_{glb}$ on P, which yields $\text{seq}(St_{glb}, T(P))$ by Lemma 1. For (b): by the specification of $update_snapshots$, the surviving entries are exactly those keyed by some $L_{con} \,\|\, K'$, re-keyed to K'. For pre-existing entries, the induction hypothesis (b) gives $St_K = \text{seq}(St_{glb}^{prev}, T(L_{con} \,\|\, K'))$, which equals $\text{seq}(St_{glb}, T(K'))$ by associativity. New entries created by the fallback rebuild satisfy the same equality by Lemma 1. Entries whose keys do not extend L_{con}—including every key that diverges from the committed order—are discarded, and $trim_window$ only removes entries. Hence (b) holds after the handler. □

The snapshot-restoration step of Fig. 5 is captured by the following contract.

Lemma 3 (Snapshot validity)**.** *Under (A2), whenever Fig. 5 invokes $find_best_snapshot$ while processing a delivery L_{con} (line 33), its return value (St_Δ, ℓ) satisfies $0 \leq \ell \leq |L_{con}|$ and*

$$St_\Delta = \text{seq}(St_{glb}^{prev}, T(L_{con}[1..\ell])),$$

where St_{glb}^{prev} is the committed state at the start of the delivery. In particular, the default return $(St_{glb}^{prev}, 0)$ is always valid, so the call is well-defined even when no stored snapshot matches.

Proof. At the invocation point, no state or snapshot store has been modified since the start of the handler, so Lemma 2 (b) holds with committed state St_{glb}^{prev}. If the call returns a stored entry (K, St_K) with $K = L_{con}[1..\ell]$ as a block sequence, then $St_K = \text{seq}(St_{glb}^{prev}, T(K)) = \text{seq}(St_{glb}^{prev}, T(L_{con}[1..\ell]))$. Otherwise, it returns $(St_{glb}^{prev}, 0)$, and $\text{seq}(St_{glb}^{prev}, []) = St_{glb}^{prev}$ by Definition 4. □

Proof of Serializability (Theorem 1). Fix a correct replica R_i. We prove by induction on the number $d \geq 0$ of consensus deliveries that the replica has processed the following invariant: $St_{glb} = \text{seq}(St_0, T^{(d)})$, where $T^{(d)}$ is the concatenation of the flattened transaction sequences of all leader blocks delivered in the first d deliveries.

Base case ($d = 0$). No deliveries have been processed; the replica's global state is the genesis state, so $St_{glb} = St_0 = \text{seq}(St_0, [])$ by Definition 4.

Inductive step. Assume the invariant holds for d, and let L_{con} denote the $(d{+}1)$-th delivery with flattened transaction sequence $T_{new} = T(L_{con})$. By the IH, the global state at the start of this delivery satisfies $St_{glb}^{prev} = \text{seq}(St_0, T^{(d)})$. By Lemma 2 (c), whose case analysis covers the direct-commit branch of Fig. 5 as well as both snapshot-and-replay cases, processing the delivery sets

$$\begin{aligned} St_{glb} &= \text{seq}(St_{glb}^{prev}, T_{new}) = \text{seq}(\text{seq}(St_0, T^{(d)}), T_{new}) \\ &= \text{seq}(St_0, T^{(d)} \,\|\, T_{new}) \end{aligned}$$

by associativity, so the invariant holds for $d{+}1$. By induction, it holds for all $d \geq 0$, which is exactly Definition 1. □

C. Proof of Linearizability

We lift the message-level guarantees of atomic broadcast (A1) to the state layer of Pufferfish. A *state transition* at correct replica R_i is one application of $execute$ whose effect is reflected in St_{glb} at R_i (i.e., persisted by a $commit_state$ call); we identify it with the transaction tx that triggered it. By Lemma 2 (c), every delivery of L_{con} sets St_{glb} to the sequential execution of $T(L_{con})$ from the previous committed state; hence every transaction in every leader block of every delivered L_{con} produces exactly one state transition, whether its $execute$ application happened speculatively (persisted at Fig. 5, line 31) or via the snapshot prefix and the re-execution loop (Fig. 5, lines 34–37, persisted at line 38). We order the state transitions of a replica by their position in this sequential schedule.

We now prove two state-layer lemmas, each lifting one BAB property. Throughout, R_i, R_j denote arbitrary correct replicas.

Lemma 4 (Same state transitions)**.** *There is no state transition performed by some correct replica that is not eventually performed by every correct replica.*

Proof. Suppose, for contradiction, that there exists a state transition τ performed by some correct replica R_i that is never performed by some correct replica R_j. Let tx be the transaction that triggers τ, contained in some L_{con} delivered to R_i. By Agreement (A1), R_j eventually delivers the same L_{con}, and by Integrity (A1) it does so once. By Lemma 2 (c), in every case of Fig. 5, processing L_{con} persists a committed state that results from applying $execute$ to every transaction in every leader block of L_{con}. Hence R_j performs the application of $execute$ on tx during its (single) processing of L_{con}, i.e., the same state transition τ, contradicting the assumption. □

Lemma 5 (Same order of state transitions)**.** *There are no state transitions τ_a, τ_b and correct replicas R_i, R_j such that R_i performs τ_a before τ_b while R_j performs τ_b before τ_a.*

Proof. Suppose, for contradiction, that there exist state transitions τ_a, τ_b and correct replicas R_i, R_j such that R_i performs τ_a before τ_b while R_j performs τ_b before τ_a. Let tx_a, tx_b be the transactions triggering τ_a, τ_b, contained in L^a_{con}, L^b_{con} respectively (as delivered to R_i). We distinguish two cases.

Case 1: $L^a_{con} = L^b_{con}$. The two transitions originate from the same delivery L_{con} at R_i. By Lemma 2 (c), processing L_{con} applies the sequential schedule $\mathrm{seq}(\cdot, T(L_{con}))$ to the previous committed state, and $T(L_{con})$ is deterministic given L_{con} and the committed leader history shared by (A1): its leader blocks are taken in L_{con}-order, and within each leader block $serialize_transactions$ produces a deterministic ordering. Hence the relative order of the state transitions on tx_a and tx_b within any processing of L_{con} is determined by that shared context. By Integrity (A1), R_j delivers L_{con} at most once and processes it once, so it performs τ_a before τ_b by the same deterministic algorithm, contradicting the assumption that R_j performs τ_b before τ_a.

Case 2: $L^a_{con} \neq L^b_{con}$. Since τ_a is performed before τ_b at R_i, the algorithm processed L^a_{con} before L^b_{con}, which by inspection of Fig. 5 requires R_i to have $a_deliver$ed L^a_{con} before L^b_{con}. By Total Order (A1), no correct replica $a_deliver$s L^b_{con} before L^a_{con}, and by Integrity (A1) R_j delivers each of L^a_{con}, L^b_{con} at most once; hence R_j delivers L^a_{con} first and processes each exactly once. By inspection of Fig. 5, τ_a is performed at R_j during its processing of L^a_{con} and τ_b during its processing of L^b_{con}, so R_j performs τ_a before τ_b, contradicting the assumption. □

Proof of Linearizability (Theorem 2). Suppose, for contradiction, that there exist correct replicas R_i, R_j and an integer n such that both $St^i_{\vdash n}$ and $St^j_{\vdash n}$ are defined and $St^i_{\vdash n} \neq St^j_{\vdash n}$.

By Lemma 4, the state transitions performed by R_i and R_j have the same support, and by Lemma 5 they are performed in the same relative order at both replicas. Hence, for every $k \leq n$, the k-th state transition at R_i applies $execute$ on the same transaction tx_k as the k-th state transition at R_j.

We show by induction on $k \in \{0, 1, \ldots, n\}$ that $St^i_{\vdash k} = St^j_{\vdash k}$. *Base case ($k = 0$):* $St^i_{\vdash 0} = St_0 = St^j_{\vdash 0}$ by the genesis convention. *Inductive step:* assuming $St^i_{\vdash k-1} = St^j_{\vdash k-1}$, both replicas apply $execute$ to the same input state on the same transaction tx_k, so by (A2) the output states agree: $St^i_{\vdash k} = St^j_{\vdash k}$.

In particular, $St^i_{\vdash n} = St^j_{\vdash n}$, contradicting the assumption. Hence, no such n exists, which is exactly Definition 2. □

D. Proof of Liveness

Liveness relies on Validity (A1). We argue that Fig. 5 terminates on every delivered L_{con} and reflects every contained transaction's execution result in St_{glb}.

Lemma 6 (Delivery)**.** *If a correct replica R_k receives a user transaction tx, then every correct replica R_i eventually $a_delivers$ L_{con} that contains a committed leader block with tx in it or in its causal history.*

Proof. By construction, a correct replica that receives a user transaction includes it in a block it broadcasts [8]–[10], and any such block (or its causal history) eventually appears in a committed leader block of some delivered L_{con}. By Validity (A1), every correct replica R_i eventually $a_deliver$s this L_{con}. □

Proof of Liveness (Theorem 3). Suppose, for contradiction, that there exists a user transaction tx submitted to a correct replica that is never executed and reflected in St_{glb} at some correct replica R_j. By Lemma 6, R_j eventually $a_deliver$s some L_{con} that contains tx in a block.

On receiving L_{con}, R_j runs Fig. 5, which terminates in finite time in every case of Lemma 2:

- each call to $speculative_execution$ (Fig. 5, lines 45–58), including the repair step at line 41, iterates over a finite prediction list, and each $execute$ invocation terminates by (A2);
- the re-execution loop (Fig. 5, lines 34–37) is bounded by $|L_{con}|$;
- the snapshot stores $\mathcal{S}_{rep}$ and $\mathcal{S}_{win}$ are finite at every handler invocation, so snapshot search and cleanup terminate, and $find_best_snapshot$ returns a valid prefix state (Lemma 3).

In every case, by Lemma 2 (c), the handler invokes $commit_state$ to set St_{glb} to the sequential execution of $T(L_{con})$ from the previous committed state. In particular, the execution result of tx is reflected in St_{glb} at R_j, contradicting the assumption.

Hence every user transaction submitted to a correct replica is eventually executed at every correct replica and reflected in St_{glb}, which is exactly Definition 3. □