

Uncoordinated Checkpointing in Stateful Transactional Systems

Decoupling Fault Tolerance from Coordination in
STYX

MSc Thesis at the Data-Intensive Systems Group

J.C. de Haan

Uncoordinated Checkpointing in Stateful Transactional Systems

Decoupling Fault Tolerance from Coordination
in STYX

by

J.C. de Haan

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science,
to be defended publicly on Thursday, August 20, 2026 at 12:30.

Student number: 4623584
Project duration: September 1, 2025 – April 1, 2026
Thesis committee: Dr. J.E.A.P. Decouchant, Data-Intensive Systems Group, chair
Dr. G. Iosifidis, Networked Systems Group
Dr. A. Katsifodimos, Data-Intensive Systems Group, supervisor

Cover: Photo by Matthieu Beaumont on Unsplash (Unsplash License)

An electronic version of this thesis is available at <https://repository.tudelft.nl/>.

Preface

I would like to thank my daily supervisor, Dr. Asterios Katsifodimos, for his guidance throughout this project, and Kyriakos Psarakis for his supervision and for building the Stryx system that this thesis extends. I would also like to thank the members of my thesis committee, Dr. J.E.A.P. Decouchant and Dr. G. Iosifidis, for their time and feedback.

*J.C. de Haan
Delft, August 2026*

Abstract

SRYX is a distributed runtime for stateful transactional functions. It executes transactions in deterministic, lockstep epochs and periodically writes state to stable storage for recovery. In the original design, each checkpoint depends on both the workers' state files and a coordinator-written global sequencer file. Consequently, a delayed or failed coordinator write can make otherwise complete worker checkpoints unusable. This thesis investigates whether checkpoint persistence can be moved from the coordinator to the workers without harming system performance or recovery.

The proposed design uses SRYX's shared epoch boundaries as consistent recovery points. Workers persist their partition checkpoints independently, while recovery selects the newest complete epoch across all partitions. Transaction execution remains coordinated; only checkpoint persistence is decentralized. The evaluation considers steady-state performance, checkpoint-file compaction, and partition rebalancing during degraded operation after a worker failure.

Across approximately 1,670 single-node runs and 334 configurations, coordinated and uncoordinated checkpoint persistence show no systematic performance difference. Their saturation points differ by -5.3% to $+6.3\%$, with no consistent winner. Recovery performance depends more strongly on how stored checkpoints are managed. Compacting every 100 checkpoints reduces state restoration time from 4,971 ms to 474 ms and provides the best tested balance between recovery speed and foreground latency. More aggressive compaction restores state faster but becomes disruptive as load increases.

Rebalancing is similarly workload-dependent. After a worker failure, the surviving workers operate with an uneven partition distribution. Redistributing those partitions reduces P95 latency from 288 ms to 146 ms at 40% utilization without affecting throughput. At loads of 50% and above, however, the newly restored worker limits epoch progress and lowers throughput during the 90-second observation window.

These results show that checkpoint persistence can be decentralized without measurable steady-state cost under the tested conditions. Compaction should balance recovery objectives against foreground load, while post-failure rebalancing should be applied selectively rather than as a universal recovery rule. The findings are limited to one workload and a single-node deployment.

Contents

Preface	i
Abstract	ii
1 Introduction	1
1.1 Context and Motivation	1
1.2 Problem Statement	2
1.3 Research Questions	2
1.4 Contributions	4
2 Background	6
2.1 STYX Architecture	6
2.1.1 Operators and Partitioning	7
2.1.2 Programming Model	7
2.2 System Model	7
2.3 The ARIA Transaction Protocol in STYX	8
2.3.1 Transaction Sequencing	8
2.3.2 Epoch Execution Pipeline	8
2.3.3 Epoch Boundaries as Consistent Cuts	10
2.4 Coordinated Checkpointing (Baseline)	11
2.4.1 Checkpoint Trigger	11
2.4.2 Delta Maps	11
2.4.3 Sequencer File Persistence	11
2.4.4 Recovery	12
2.4.5 Compaction	12
2.5 Exactly-Once Processing Guarantee	12
2.6 Summary	13
3 Related Work	14
3.1 Checkpointing in Distributed Systems	14
3.1.1 Coordinated Checkpointing	14
3.1.2 Uncoordinated Checkpointing	14
3.1.3 Checkpointing in Stream Processing	14
3.2 Transactional Stateful Functions Platforms	15
3.2.1 Serializable SFaaS Systems	15
3.2.2 Weaker Consistency Models	15
3.3 Deterministic Database Protocols	15
3.3.1 Calvin and ARIA	15
3.3.2 Other Deterministic Protocols	16
3.4 State Migration and Elasticity	16

3.5	LSM-Tree Compaction	16
3.6	Summary and Research Gap	17
4	Design and Implementation	18
4.1	Uncoordinated Checkpointing with Epoch Alignment	18
4.1.1	Motivation	18
4.1.2	Epoch-Aligned Time-Based Strategy	19
4.1.3	Trade-offs and Recovery Line Bounds	20
4.2	Per-Partition Sequencer Files	20
4.2.1	Design	20
4.2.2	Storage Format	20
4.2.3	Write Order Invariant	21
4.2.4	Recovery with Per-Partition Sequencer Files	21
4.2.5	Correctness Argument	22
4.3	Epoch-Delta Compaction	22
4.3.1	Problem: Delta File Accumulation	22
4.3.2	From Coordinator-Driven to Worker-Driven Compaction	22
4.3.3	Compaction Algorithm	23
4.3.4	Scoped Compaction	24
4.3.5	Sequencer File Compaction	24
4.3.6	Interaction with Recovery Line Computation	24
4.3.7	Compaction Trigger Strategies	25
4.4	Failure Detection and Recovery	25
4.4.1	Failure Detection	25
4.4.2	Recovery Process	26
4.4.3	Recovery Epoch Computation from Stable Storage	26
4.4.4	In-Progress Epochs During Recovery	26
4.4.5	End-to-End Recovery Correctness	27
4.5	Partition Rebalancing	27
4.5.1	Motivation	27
4.5.2	Rebalance Recovery Protocol	28
4.5.3	Correctness	30
4.5.4	Compaction-Rebalance Interaction	31
4.6	Differences Between Coordinated and Uncoordinated Modes	31
5	Experiment Design	32
5.1	Methodology	32
5.1.1	General Principles	32
5.1.2	Hardware and Software Environment	32
5.1.3	Workload	32
5.1.4	Metrics	33
5.2	Cluster Configuration	34
5.3	Experiment Suite 1: Scalability (RQ1a–RQ1d)	34
5.3.1	Phase 1: Saturation Point Discovery (RQ1a, RQ1c)	34
5.3.2	Phase 2: Epoch Size Impact (RQ1b)	35
5.3.3	Phase 3: Checkpoint Strategy Sweep (RQ1c)	35

5.3.4	Phase 4: Contention Sweep (RQ1d)	35
5.3.5	Experiment Matrix Summary	36
5.4	Experiment Suite 2: Compaction and Recovery (RQ2a–RQ2d)	36
5.4.1	Phase 1: Steady-State Overhead	36
5.4.2	Phase 2: Recovery Performance	36
5.4.3	Phase 3: Load-Dependent Compaction Overhead	37
5.4.4	Experiment Matrix Summary	37
5.4.5	Analysis Plan	37
5.5	Experiment Suite 3: Rebalance Recovery (RQ3a–RQ3b)	38
5.5.1	Common Configuration	38
5.5.2	Phase 1: Rebalance Overhead (RQ3a)	38
5.5.3	Phase 2: Rebalance Benefit (RQ3b)	39
5.5.4	Experiment Matrix Summary	41
5.6	Scalability Analysis Framework	41
5.6.1	Amdahl’s Law Fitting	41
5.6.2	Statistical Reporting	41
5.7	Threats to Validity	41
6	Results	43
6.1	Scalability Results (RQ1a–RQ1d)	43
6.1.1	Phase 1: Saturation Points	43
6.1.2	Scalability Analysis (RQ1a)	44
6.1.3	Phase 2: Epoch Size Impact (RQ1b)	46
6.1.4	Phase 3: Checkpoint Strategy Sweep (RQ1c)	47
6.1.5	Phase 4: Contention Impact (RQ1d)	49
6.2	Compaction and Recovery Results (RQ2a–RQ2d)	50
6.2.1	State Restoration Time by Configuration	50
6.2.2	File Count–Restoration Time Correlation (RQ2a)	51
6.2.3	I/O vs. CPU Breakdown (RQ2b)	52
6.2.4	Compaction Interval Trade-off (RQ2c)	52
6.2.5	Load-Dependent Compaction Overhead (RQ2c cont.)	53
6.2.6	Continuous vs. Recovery-Only Compaction (RQ2d)	55
6.2.7	Two-Phase Recovery and End-to-End Timeline	56
6.2.8	Discussion	56
6.2.9	Consistency Verification	57
6.3	Rebalance Recovery Results (RQ3a–RQ3b)	58
6.3.1	Rebalancing Overhead (RQ3a)	58
6.3.2	Latency and Capacity Benefit (RQ3b)	58
6.3.3	Consistency Verification	60
6.3.4	Limitation: High-Ratio Failures	60
6.4	Summary	60
7	Conclusion and Future Work	61
7.1	Summary	61
7.2	Answers to the Research Questions	61
7.2.1	RQ1: Steady-State Performance	61

7.2.2	RQ2: Compaction and Recovery	62
7.2.3	RQ3: Rebalancing After Recovery	62
7.3	Limitations	63
7.4	Future Work	63
7.4.1	Adaptive Compaction Scheduling	63
7.4.2	Diagnosing and Reducing the Cold-Worker Penalty	63
7.4.3	Long-Duration Rebalance Evaluation	63
7.4.4	High-Ratio Failure Recovery	63
7.4.5	Validation Under Extreme Contention	63
7.4.6	Distributed Deployment	64
7.5	Closing Remarks	64
A	Supplementary Results	65
A.1	Compaction Overhead as Percentages	65
B	Implementation File Map	66
	References	67

List of Figures

1.1	Coordinated vs. uncoordinated checkpointing	1
2.1	One epoch under lockstep execution	10
4.1	The recovery line over per-partition checkpoints	22
4.2	Epoch-delta compaction	23
4.3	Failure-to-resume sequence	25
4.4	Recovery-point sources in the two modes	26
4.5	Selective restore during rebalance	28
6.1	Offered input vs. committed throughput	44
6.2	Saturation throughput by mode	45
6.3	Observed speedup vs. Amdahl's Law fit	46
6.4	P95 latency vs. epoch size	47
6.5	Restoration time vs. files fetched	51
6.6	Restoration timing breakdown	52
6.7	Steady-state P95 by compaction configuration	53
6.8	Restoration time vs. steady-state P95	54
6.9	P95 latency vs. utilization by interval	55
6.10	Measured recovery timeline and state restoration	57

List of Tables

2.1	Notation and parameters	6
3.1	Positioning relative to related systems	17
4.1	Coordinated vs. uncoordinated modes	31
5.1	Experiment server specification.	33
5.2	Default cluster configuration.	34
5.3	Scalability experiment matrix.	36
5.4	Compaction experiment matrix (all phases).	37
5.5	Load levels and survivor utilization	40
5.6	Rebalance experiment matrix ($w = 4$, saturation TPS = 18,000).	41
6.1	Saturation throughput by worker count and mode	43
6.2	Scaling efficiency relative to $w = 2$	44
6.3	Epoch size impact on P95 latency	46
6.4	Checkpoint strategy sweep at $w = 2$	48
6.5	Checkpoint strategy sweep across worker counts	48
6.6	Committed throughput under Zipfian skew	49
6.7	P95 latency by contention level	50
6.8	Restoration time by compaction configuration	51
6.9	Restoration timing breakdown by phase	52
6.10	Restoration time vs. steady-state overhead	52
6.11	P95 latency by load and compaction interval	54
6.12	Continuous vs. recovery-only compaction	55
6.13	Two-phase recovery breakdown	56
6.14	Rebalance timing breakdown	58
6.15	Post-failure throughput and latency by load	59
A.1	P95 overhead relative to $C = 0$	65
B.1	Key implementation files for each contribution.	66

Introduction

1.1. Context and Motivation

This thesis studies whether a transaction-processing system can recover from crashes without routing every checkpoint through a central coordinator. Figure 1.1 compares the two designs.

Cloud applications such as payment processors, reservation platforms, and inventory systems need fast transaction processing with strong consistency. Stateful Functions-as-a-Service (SFaaS) combines serverless scaling with streaming state management to meet these needs [28, 13, 37].

SRYX [28] is a dataflow-based SFaaS runtime for serializable, exactly-once transactions over stateful functions. It extends the deterministic ARIA protocol [25] to support arbitrary function call graphs on streaming dataflows. The original evaluation reports at least an order of magnitude more throughput than comparable serializable SFaaS systems, including Boki, Beldi, and T-Statefun [28, 19, 36, 16].

SRYX combines periodic asynchronous checkpoints in durable storage with deterministic replay from Apache Kafka. The original system uses *coordinated checkpointing* (Section 2.4). A checkpoint becomes usable only after every worker has saved its state and the coordinator has collected their snapshot identifiers, selected the minimum, and written a global sequencer file. Recovery therefore depends on all worker writes plus one coordinator write.

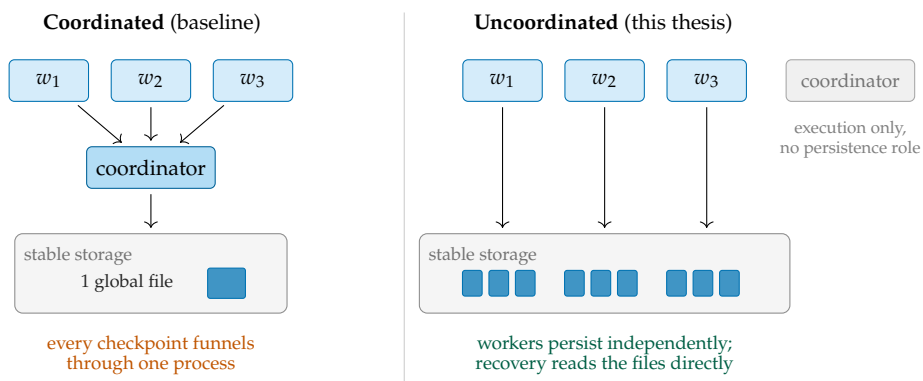


Figure 1.1: Checkpoint persistence in the baseline and redesigned systems. Left: workers route checkpoint metadata through the coordinator, which writes one global file per checkpoint. Right: each worker writes per-partition files directly to stable storage. The coordinator continues to orchestrate execution but does not persist checkpoints.

Uncoordinated checkpointing does not remove persistence failures in general. If a worker cannot save one or more partitions, the common recovery point still stalls. A shared-storage failure also affects both modes. The redesign removes only the extra coordinator dependency: workers save their own state and metadata, so a failed coordinator write cannot block checkpoints that workers have already stored.

In the coordinated system, workers continue processing during such a fault while the recovery point grows older. A later worker failure must recover from an older checkpoint and replay more work. The measured evaluation does not inject this fault; it measures the performance cost of removing the dependency.

A separate recorded demonstration¹ keeps worker writes healthy while making coordinator writes fail. Under the same load, the coordinated recovery point stops for minutes while the uncoordinated recovery point advances. After a worker failure, the coordinated cluster restores from before the injected fault, while the uncoordinated cluster restores state written seconds earlier. Both produce exactly-once output. The coordinator still runs every execution barrier in both modes, and coordinator fault tolerance remains outside our scope (Section 2.2). Execution stays coordinated; only checkpoint persistence becomes uncoordinated.

We therefore study *uncoordinated checkpointing*: each worker decides when to checkpoint and writes its state and sequencer metadata directly to stable storage. “Uncoordinated” refers only to checkpoint scheduling and persistence; transaction execution still uses ARIA’s epoch barriers (Section 2.3). Prior work covers uncoordinated checkpointing [11] and deterministic recovery from a replayable input log [35, 30]. To our knowledge, prior work has not combined these ideas for deterministic, epoch-based transactions over streaming dataflows, where lockstep boundaries already create consistent cuts.

1.2. Problem Statement

Moving checkpoint writes to the workers shifts several responsibilities. Workers may save different epochs, so recovery must find the newest epoch available for every partition. If no recent epoch is shared, the system must recover from an older point and replay more transactions. Per-partition delta files also accumulate over time and can slow state restoration unless they are compacted. After a failure, surviving workers take over the failed worker’s partitions. When that worker returns, rebalancing is useful only if the restored capacity justifies the cost of pausing the system and moving the partitions. We therefore evaluate whether the design preserves steady-state performance, keeps state restoration efficient, and improves performance after a failure.

1.3. Research Questions

This thesis addresses the following research questions.

RQ1: How does uncoordinated checkpointing compare to coordinated checkpointing in terms of steady-state performance?

Worker count, epoch size, checkpoint strategy, and contention may each interact with the checkpointing mode. We therefore ask four sub-questions:

RQ1a: *How do throughput and scaling efficiency change with worker count across the tested range?*

RQ1b: *Which epoch size provides the best throughput-latency trade-off among the tested configurations?*

RQ1c: *Does the checkpointing protocol affect steady-state performance?*

¹The reproducible script, recording, and recorded results are available in `demo/demo-coordinator-stall/`.

RQ1d: *How does key contention degrade performance, and does increasing worker count worsen the effect?*

RQ2: What is the effect of epoch-delta compaction on state restoration time, and which compaction strategy performs best among the tested configurations?

Compaction merges accumulated delta files into a single base file. We decompose this into four sub-questions:

RQ2a: *Does reducing delta file count reduce state restoration time?*

RQ2b: *Is recovery I/O-bound or CPU-bound?*

RQ2c: *Which compaction interval performs best among the tested configurations?*

RQ2d: *Does recovery-only compaction provide the same benefit as continuous compaction?*

RQ3: Does post-failure partition rebalancing restore system capacity, and what is the associated overhead?

We compare the cost of rebalancing with the capacity it restores through two sub-questions:

RQ3a: *How much overhead does rebalancing add?*

RQ3b: *Does rebalancing provide a sustained performance benefit?*

1.4. Contributions

This thesis makes the following contributions:

1. **Uncoordinated checkpointing with two trigger strategies.** In a lockstep, deterministic system, every epoch boundary is already consistent. Checkpointing therefore needs no extra coordination messages, and recovery only needs to find the latest checkpoint epoch available for every partition. We implement two triggers: a pure epoch-based approach, which checkpoints every N epochs without using local clocks, and an epoch-aligned time-based approach, which uses multiples of a configurable interval A to approximate a wall-clock schedule. We also describe when the time-based approach can miss a shared epoch (Section 4.1.3).
2. **Per-partition sequencer files.** In coordinated checkpointing, the coordinator collects checkpoint metadata and writes one global sequencer file. We replace this with per-partition files written directly by workers. Each file records the Kafka offsets, epoch counter, and transaction counter needed to recover that partition. Because workers synchronize these counters at every epoch boundary, recovery can use the files without coordinator-written metadata. Workers still send SnapID notifications for monitoring, but checkpoint durability and progress do not depend on them.
3. **Worker-driven compaction with configurable strategies.** Compaction must preserve at least one epoch shared by all partitions. Each worker therefore compacts only its own partitions, avoiding concurrent writes, and leaves recent delta files uncompact. We support periodic compaction at interval C and recovery-only compaction instead of always compacting after each snapshot.
4. **Partition rebalancing after recovery.** A rejoined worker can reclaim partitions through a pause-redistribute-restore protocol that reuses the existing recovery path. The tested runs show no consistency violations. Rebalancing improves tail latency at low load, but at loads of $\sim 50\%$ and above it reduces throughput because every ARIA barrier waits for the newly restored, slower worker. Restoring the partition count therefore does not immediately restore performance. The observed slowdown motivates warm-up-aware rejoining and live state transfer (Section 7.4.2).

5. **Empirical evaluation.** We run $\sim 1,670$ experiments across ~ 334 configurations, usually with 5 repetitions, and report medians. Across six worker counts, reducing SRYX’s default epoch size from 1,000 to 100 cuts median P95 latency by 13–14 $\times$ in both modes. State restoration time also follows a linear per-file model in the tested storage environment, suggesting higher costs when per-object I/O is slower. All tested configurations with Zipfian skew $z \leq 0.75$ pass exactly-once verification. At $z = 0.99$, the uniform-workload saturation rate overloads both modes, so some runs do not finish within the validation window; these runs show a capacity limit, not a demonstrated correctness failure.

2

Background

This chapter introduces **Stryx**, states the system assumptions, explains how **ARIA** executes transactions, and describes the coordinated checkpointing baseline. These details provide the terms and invariants used in the design.

We use the notation in Table 2.1 throughout the thesis.

Table 2.1: Notation and default parameter values. Subscripted E_i denotes the i -th epoch (an identifier), distinct from the epoch size E .

Symbol	Meaning	Default
n / w	Number of workers (total / varied in experiments)	2–12
f	Number of simultaneous worker failures	1
E	Epoch size (transactions per batch)	1,000
N	Checkpoint interval (epochs between checkpoints)	N/A
C	Compaction interval (checkpoints between compactions)	0 (off); 100 (best tested)
A	Alignment interval (epochs) for time-based triggers	50
T	Target wall-clock interval between checkpoints	10 s
d	Average epoch duration	≈ 100 ms
B	Minimum compaction cutoff margin (epochs)	100
r	Epoch rate at saturation (per-worker committed TPS / E)	≈ 4.5 –6/s
z	Zipfian skew parameter (key contention)	0–0.99
S	Effective non-scaling fraction (fitted)	$\approx 22\%$

2.1. **stryx** Architecture

Stryx [28] is a distributed dataflow-based runtime for executing transactional stateful functions. Its architecture has four components:

Coordinator. Manages the cluster by registering workers, distributing operator partitions, monitoring health via heartbeats, and orchestrating synchronization barriers during transaction processing. The coordinator does not process transactions directly.

Workers. The primary processing units. Each worker hosts a *sequencer* coroutine that assigns transaction identifiers to incoming requests, and a *function executor* coroutine that processes the sequenced transactions. Workers maintain operator state in memory, partitioned by key.

Apache Kafka [21]. Kafka carries both client requests and transaction results. Offset-based replay enables deterministic recovery. The input log must retain records back to the current recovery line. Otherwise, replay starts at the earliest retained offset and silently skips older transactions. Deployments should therefore compare recovery-line age with topic retention. Restarting from empty state (Section 4.2) requires retention back to the initial offsets; bounded-retention deployments must instead guarantee a non-empty checkpoint intersection.

Stable storage (MinIO/S3). Provides durable storage for checkpoints. Workers persist incremental state snapshots as binary files, organized by operator, partition, and epoch.

2.1.1. Operators and Partitioning

State in Stryx is organized into *operators*, each representing a logical entity type (e.g., `BankAccount`, `Hotel`). Each operator is partitioned across workers using hash partitioning on entity keys. A worker may host partitions from multiple operators.

Invariant 2.1 (Partition Ownership). *At any point in time, each operator partition is owned by exactly one worker.*

The coordinator’s partition assignment maintains this invariant, preventing concurrent modifications to the same partition.

2.1.2. Programming Model

Stryx provides a Python API for defining operators and functions. A function accesses local state through a context object and can call functions on other operators, including remote ones. An external request starts a transaction, whose scope is the resulting function call graph.

2.2. System Model

We assume the following system model throughout this thesis:

Failure model. We assume *crash recovery*: a process may crash and later restart. We exclude Byzantine failures [23] and assume the coordinator remains available. We also assume that no second failure occurs during recovery. This limits liveness rather than safety: a second failure would stall progress until it is detected, but workers would not resume in the meantime. Finally, a worker marked as failed must stop writing to stable storage. Crashes satisfy this condition; network partitions and false suspicions would require write fencing, which we do not implement.

Simultaneous failures. In principle, the system tolerates any $f < n$ worker failures if the coordinator and object store remain available. Stable storage, not worker replication, holds the authoritative checkpoints. At least one worker must survive to serve all partitions. We test only $f = 1$ with $n = 4$. In the current implementation, failures of at least half the workers exceed the coordinator’s recovery timeout (Section 6.3.4).

Communication model. Communication between workers, and between workers and the coordinator, is asynchronous and reliable (messages are eventually delivered if both endpoints are alive). The ARIA barriers use a gather-broadcast pattern through the coordinator, which acts as a synchronization point. Assumptions on input-log retention are stated with the Kafka component above.

Durable storage. Stable storage (MinIO, an S3-compatible object store, in our deployment) is assumed to be always available and to provide strong read-after-write consistency, including for in-place overwrites (relied upon by sequencer-file compaction, which rewrites a file under a fixed key). Modern S3 provides this guarantee [2] for both new objects and overwrites. Checkpoint files written to stable storage are durable and globally visible once the write completes.

2.3. The ARIA Transaction Protocol in STYX

STYX adapts the deterministic ARIA OLTP protocol [25] to stateful functions. It processes transactions concurrently in batches called *epochs*. The *epoch size* is the number of transactions in one batch (1,000 by default). An epoch used as a count or index refers to the batch counter. Thus, “every 100 epochs” means every 100 batches, not every 100 transactions.

2.3.1. Transaction Sequencing

Each worker assigns globally unique transaction identifiers (TIDs) without contacting other sequencers. Worker 1 takes IDs $1, n+1, 2n+1, \dots$; worker 2 takes $2, n+2, 2n+2, \dots$; and so on. Each worker therefore has its own sequence of IDs. Formally:

$$\text{TID}_{sid,lc} = sid + (lc \times n_{\text{seq}}) \quad (2.1)$$

Here, $sid \in \{1, \dots, n_{\text{seq}}\}$ is the sequencer identifier, a contiguous index derived from sorting registered worker IDs. The counter $lc \in \mathbb{N}_0$ is the local transaction counter (denoted `t_counter` in the implementation), incremented per TID assignment and synchronized to the global maximum at each epoch boundary (Invariant 2.2). The modulus $n_{\text{seq}} \in \mathbb{N}_{\geq 1}$ is the total number of sequencers (equal to workers). This round-robin assignment is similar to Mencius [26].

2.3.2. Epoch Execution Pipeline

During an epoch, workers first run all transactions in parallel on a snapshot of the data. They then meet at a *barrier* to identify conflicts and decide which transactions must run again. Finally, they apply accepted changes and return results. At a barrier, every worker waits until all others arrive; this keeps execution in lockstep.

Each epoch has three mandatory synchronization barriers, all orchestrated by the coordinator in a gather-broadcast pattern (Figure 2.1). Some configurations add more. When ARIA’s transaction-reordering optimization is enabled, `DeterministicReordering` runs between `AriaProcessingDone` and `AriaCommit` and reorders conflicting transactions instead of aborting them. We disable this optimization in all experiments. The fallback phase adds `AriaFallbackStart` and `AriaFallbackDone`. The next paragraphs describe the three mandatory barriers:

Phase 1: Parallel execution. All transactions in the epoch execute concurrently on a snapshot of the current state. During execution, each worker records read/write sets for conflict detection. Cross-worker function calls are dispatched via TCP channels, and function completion is tracked using the *acknowledgment share* mechanism [28]. The root function’s acknowledgment is split into fractional shares distributed across the call graph, and the transaction completes when all shares sum to 1.

Barrier 1: `AriaProcessingDone`. After all transactions in the epoch finish execution, workers exchange *logic aborts*, that is, transactions that failed due to application-level exceptions (e.g., insufficient funds).

The coordinator aggregates the union of all workers' logic abort sets and broadcasts the result. After this barrier, all workers agree on which transactions have failed before conflict detection begins.

Phase 2: Conflict detection. Each worker analyzes its local read/write sets to detect conflicts. A conflict occurs when two transactions access the same key and at least one access is a write. Following ARIA's protocol, the transaction with the lower TID always wins; the higher-TID transaction is aborted and deferred to the fallback phase or next epoch.

Barrier 2: AriaCommit. Workers exchange three pieces of information: (1) locally detected concurrency aborts, (2) the current $t_counter$, and (3) the epoch's sequence size. The coordinator computes:

- $global_aborts = \bigcup_{w \in W} local_aborts(w)$
- $max_t_counter = \max_{w \in W} t_counter(w)$

and broadcasts both values to all workers. This is the most critical synchronization point:

Invariant 2.2 ($t_counter$ Synchronization). *At every epoch boundary e :*

$$\forall w_i, w_j \in W : w_i.t_counter = w_j.t_counter = max_t_counter(e)$$

This invariant mitigates sequence imbalance [28]. Without synchronization, a busy worker's $t_counter$ would grow faster over successive epochs, producing higher TIDs and thus systematically losing conflict resolution against less-busy workers (since lower TIDs always win). Setting all workers to the global maximum rebalances transaction priorities each epoch, reducing tail latency under skewed workloads.

Phase 3: Commit. Workers apply the write sets of non-aborted transactions to operator state and send responses to clients through the output Kafka topic. A fallback phase re-executes conflicting transactions under deterministic per-key locking in TID order while non-conflicting transactions continue concurrently. The setting `FALLBACK_STRATEGY_PERCENTAGE` controls when this phase runs. Its shipped default of `-0.1` enables fallback in every non-empty epoch, regardless of the abort rate. We retain this default, so every measured epoch executes the three mandatory barriers plus the two fallback barriers, for five barriers in total.

Post-commit processing. After the commit phase, each worker increments its epoch counter, performs epoch cleanup (clearing abort sets, resetting internal buffers), and, if a checkpoint is due, captures an incremental snapshot for persistence to stable storage; the upload completes asynchronously (Section 4.1). These operations occur locally before the final synchronization barrier.

Barrier 3: SyncCleanup. Workers exchange telemetry metrics and confirm that all post-commit processing (including epoch counter increment, cleanup, and checkpoint capture) is complete. No worker may begin processing the next epoch's transactions until all workers have passed this barrier.

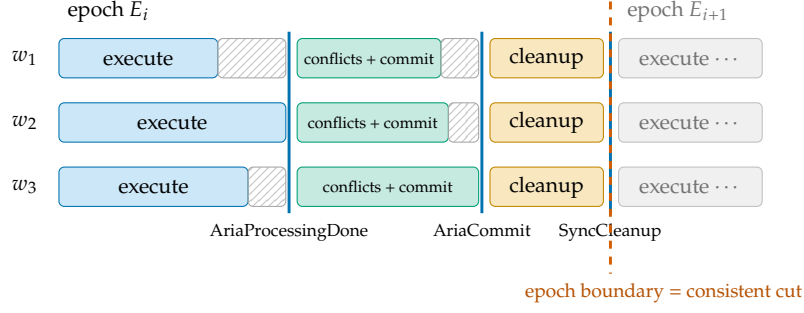


Figure 2.1: One epoch under lockstep execution. Each barrier (vertical lines) is a gather-broadcast synchronization: no worker proceeds until all have reported, so the slowest worker paces the epoch (hatched segments are waiting time). The SyncCleanup boundary is the consistent cut at which checkpoints are taken (Proposition 2.1).

Invariant 2.3 (Lockstep Execution). *Workers execute epochs in lockstep: all workers process the same epoch simultaneously. Between consecutive barriers, workers may be at slightly different progress points, but every gather-broadcast barrier forces synchronization before any worker may proceed. No worker begins epoch E_{i+1} until all workers have passed the SyncCleanup barrier for E_i :*

$$\forall w_i, w_j \in W : w_i.\text{epoch} = w_j.\text{epoch}$$

Here W is the coordinator’s active-worker set; when recovery after a failure completes in uncoordinated mode, the coordinator resets barrier membership to the surviving workers (Section 4.4).

2.3.3. Epoch Boundaries as Consistent Cuts

A *consistent cut* partitions a distributed computation into a “before” and “after” such that no causal dependency crosses from after to before [7]. Intuitively, a consistent cut is a snapshot that “could really have happened”: a photograph of the whole system in which you never see an effect without its cause. If the photo shows money arriving in account B , it must also show that money leaving account A ; a snapshot capturing the arrival but not the departure is inconsistent and could never have occurred in reality.

Our checkpointing design relies on every epoch boundary in STYX being a *consistent global cut* in the sense of Chandy and Lamport [7]:

Proposition 2.1 (Epoch Boundary Consistency). *Every epoch boundary between E_i and E_{i+1} in STYX is a consistent global snapshot, satisfying:*

1. *All workers have completed processing E_i .*
2. *No transaction from E_i has system-internal causal dependencies on transactions in E_{i+1} .*
3. *All workers agree on the committed and aborted transaction sets for E_i .*
4. *All state modifications from committed transactions in E_i are reflected in the workers’ local state.*

Proof sketch. Properties (1) and (3) follow from the SyncCleanup and AriaCommit barriers, respectively: neither lets a worker proceed until all workers have reported. With the always-on fallback pair, final agreement on the commit set occurs at AriaFallbackDone, still before the epoch boundary. Property (2) follows from the same barrier structure. No worker begins epoch E_{i+1} until all workers have committed E_i , so a transaction in E_i cannot read state produced by E_{i+1} ; causal dependencies cross epoch boundaries

only in the forward direction. Property (4) follows because the commit phase applies writes before SyncCleanup. $\square$

Why uncoordinated checkpointing is normally hard: the domino effect. In classical uncoordinated checkpointing, independent snapshots can create inconsistent message histories and cascading roll-backs (the domino effect; Chapter 3). Stryx avoids this because the epoch barrier drains intra-epoch communication: every cross-worker call issued in E_i is acknowledged and its effects applied before the boundary releases. Snapshots at that boundary therefore agree, with no channel state to record.

Two properties are easily conflated here. *Snapshot consistency* and transaction *isolation* are orthogonal, and Stryx obtains them from different mechanisms. The epoch barrier provides the consistent cut (snapshot consistency), while ARIA's conflict detection and deterministic commit provide serializability (isolation). Because the barrier already exists for transaction processing, Stryx inherits the consistent cut without checkpoint-specific coordination messages; under the tested single-node conditions, this produced no measurable steady-state cost.

2.4. Coordinated Checkpointing (Baseline)

The original Stryx system implements coordinated checkpointing, which we use as the baseline for comparison.

2.4.1. Checkpoint Trigger

Each worker maintains a local timer. When the elapsed time since the last checkpoint exceeds a configurable interval (default: 10 seconds), the worker takes a checkpoint at the end of the current epoch and sends the coordinator a SnapID message with the snapshot timing, Kafka offsets, and epoch counter [28]. Each timer fires independently, but the ARIA barriers keep workers in lockstep, so their trigger epochs remain close. The coordinator tolerates any residual drift by choosing the minimum completed snapshot identifier.

2.4.2. Delta Maps

Stryx uses incremental snapshots based on *delta maps*. Each operator partition maintains a delta map that tracks state mutations since the last checkpoint. When a checkpoint is triggered:

1. The delta map contents are serialized and persisted to stable storage as a binary file at path `data/{operator}/{partition}/{snapshot_id}.bin`.
2. The delta map is cleared for the next interval.

This is efficient because only modified keys are persisted rather than the entire state. The cost of a checkpoint is proportional to the number of modified keys: specifically, $|\Delta| + c$ where $|\Delta|$ is the number of modified keys since the last checkpoint and c is a small constant for metadata (offsets, counters) [28].

2.4.3. Sequencer File Persistence

In coordinated mode, the coordinator writes a *global sequencer file* that contains the merged Kafka input/output offsets and sequencer counters from all workers:

$$\text{sequencer}/\{\text{snapshot_id}\}.bin \leftarrow \{\text{input_offsets}, \text{output_offsets}, \text{epoch}, \text{t_counter}\}$$

The coordinator writes this file once the minimum snapshot identifier across all workers advances, that is, once all workers have completed that numbered checkpoint (possibly at slightly different epochs). This places the coordinator on the checkpoint persistence critical path.

2.4.4. Recovery

Upon detecting a worker failure, the coordinator:

1. Determines the recovery snapshot as $\min_{w \in W}(\text{snapshot_id}(w))$.
2. Instructs all workers to load the global sequencer file for that snapshot identifier and restore their operator state by merging each partition's delta files up to the epoch recorded in it.
3. Resets Kafka consumer offsets to the positions recorded in the sequencer file.
4. Resumes processing. Deterministic replay from the Kafka offsets re-executes all transactions between the recovery epoch and the failure point.

2.4.5. Compaction

Accumulated delta files are periodically merged (*compacted*) into a single consolidated file per partition. After each completed cluster snapshot, the coordinator triggers a background compaction process. This process loads all delta files for each partition in epoch order, merges them using dictionary update semantics (later values overwrite earlier ones), and writes the result as a new base file. This reduces the number of files to load during recovery and reclaims storage. In coordinated mode, compaction always runs after every snapshot, with no configurable compaction interval.

2.5. Exactly-Once Processing Guarantee

Stryx's exactly-once guarantee relies on the combination of four properties:

1. **Deterministic execution.** Given the same initial state and the same input sequence, Stryx's workers produce the same per-transaction outcomes and outputs, and hence the same final state. This follows from the deterministic TID assignment (Equation (2.1)) and the epoch lockstep execution (Invariant 2.3). The `t_counter` synchronization (Invariant 2.2) further ensures that all workers share the same counter value, which simplifies recovery (any partition's sequencer file contains the correct starting `t_counter`).
2. **Replayable input.** Apache Kafka allows consuming messages from a specific offset, enabling replay of the exact input sequence after recovery.
3. **Consistent checkpoints.** Checkpoints are taken at epoch boundaries (Proposition 2.1), capturing a state that includes all committed transactions up to that epoch and no partial results.
4. **Output deduplication.** Each checkpoint records its Kafka output offset. During recovery, a worker reads forward from that offset to find results already produced after the checkpoint and suppresses them during replay. This assumes the output path neither loses nor duplicates acknowledged records. Our single-node deployment uses `acks=1` without producer idempotence. Production deployments should use `acks=all` and an idempotent producer to handle broker failover and retries.

Together, these properties restore the state that failure-free execution would have produced, with each transaction applied exactly once (Lemma 7.1 in [28]). That lemma covers coordinated recovery. Section 4.4.5 gives the corresponding argument for uncoordinated recovery. The experiments verify exactly-once behavior for Zipfian skew $z \leq 0.75$; the $z = 0.99$ runs do not reach quiescence and are inconclusive for correctness (Section 6.1.5).

2.6. Summary

Three existing properties enable uncoordinated checkpointing: epoch boundaries are consistent global cuts, every worker has the same `t_counter` at those boundaries, and deterministic execution supports recovery by replay. The next chapter relates these properties to prior checkpointing and transaction-processing systems.

3

Related Work

3.1. Checkpointing in Distributed Systems

Checkpointing and rollback recovery are well studied in message-passing systems. Elnozahy et al. [11] divide checkpoint protocols into three families: *coordinated*, *uncoordinated*, and *communication-induced*. The last is a hybrid that forces checkpoints using information attached to application messages, so it does not fit Stryx’s barrier-based model.

3.1.1. Coordinated Checkpointing

In coordinated checkpointing, processes synchronize to create a consistent global checkpoint. Chandy and Lamport [7] proposed a marker-based algorithm for capturing such snapshots. Modern stream processors, including Apache Flink [6, 5], use variants of this approach: a coordinator inserts barriers into the stream, and each operator saves local state when it receives one. Recovery is simple because every completed checkpoint is consistent. However, completion waits for the slowest operator, and multi-input operators must align barriers across channels. Backpressure can therefore delay checkpoints.

3.1.2. Uncoordinated Checkpointing

In uncoordinated checkpointing, each process saves state independently. This avoids global checkpoint coordination but can cause the *domino effect* [11]: rolling back one process may invalidate other checkpoints through message dependencies, causing further rollbacks. Such protocols usually attach dependency information to application messages to find a safe recovery line.

The main challenge is finding the *recovery line*: the latest consistent global state. General message-passing systems must analyze dependencies among checkpoints and messages, which can be expensive [11].

3.1.3. Checkpointing in Stream Processing

Siachamis et al. [31] present CheckMate, a framework for evaluating checkpointing protocols in streaming dataflows. Their work compares coordinated and uncoordinated approaches in continuous stream processing, measuring checkpoint overhead, recovery time, and state consistency. Silvestre et al. [32] propose Clonos, which achieves consistent causal recovery in highly-available streaming dataflows

through causal logging and deterministic replay.

Stryx differs because deterministic epoch execution creates a consistent cut at every epoch boundary (Chapter 2). This prevents the domino effect without message logging. Classical rollback protocols combine independent checkpoints with message logs so each process can replay beyond the recovery line [11]. Stryx instead replays deterministic input from Kafka. Its existing ARIA barriers drain inter-worker messages before each epoch boundary, so checkpointing can use those boundaries without adding another barrier.

3.2. Transactional Stateful Functions Platforms

SFaaS systems differ in the consistency guarantees they provide. We separate systems with serializable transactions from those with weaker guarantees.

3.2.1. Serializable SFaaS Systems

Beldi [36] implements fault-tolerant transactional stateful serverless workflows on AWS Lambda using linked distributed atomic affinity logging on DynamoDB. It uses a variant of two-phase commit (2PC) to coordinate cross-function transactions. Logging each function execution adds one round-trip per function call.

Boki [19] extends Beldi with a shared log that improves locking and workflow re-execution. Halfmoon [29] further extends Boki with a theoretically minimal fault-tolerance logging protocol.

T-Statefun [16] builds on Apache Flink’s StateFun framework¹, adding 2PC with coordinator functions to support serializable transactions. It relies on Flink’s RocksDB [10] state backend for incremental checkpointing.

Beldi, Boki, and T-Statefun separate state from computation and log every function call, which adds latency. Stryx instead co-locates state and computation and checkpoints at epoch boundaries. Our work removes the coordinator from those checkpoint writes.

3.2.2. Weaker Consistency Models

Cloudburst [33] provides causal consistency within DAG workflows but does not guarantee serializability. **Netherite** [4] offers exactly-once guarantees for Microsoft’s Durable Functions but does not provide cross-function transactional serializability. Neither provides cross-function serializable transactions, making direct performance comparison uninformative.

3.3. Deterministic Database Protocols

Deterministic database systems guarantee that a given transaction order produces the same final state regardless of thread scheduling or failures [1, 34]. Stryx follows this model.

3.3.1. Calvin and ARIA

Calvin [35] is a deterministic database that uses a partitioned sequencer to establish a global transaction order, requiring $O(n^2)$ messages per batch between sequencers and the schedulers of all partitions. Stryx adopts a method similar to Mencius [26] that achieves $O(1)$ *sequencer* communication by assigning unique transaction identifiers based on worker ID and a local counter. This applies to transaction-ID assignment only; the per-epoch barrier synchronization described in Chapter 2 remains, so Stryx removes

¹<https://nightlies.apache.org/flink/flink-statefun-docs-stable/>

the sequencer’s quadratic cost rather than all coordination.

Calvin shows that determinism and a replayable input log remove the need to coordinate checkpoints with a physical redo log: recovery replays input from the last checkpoint [35]. CALC [30] takes asynchronous checkpoints at *virtual points of consistency* without stopping a main-memory database. Stryx’s epoch boundaries serve the same purpose in a streaming dataflow.

ARIA [25] is the deterministic OLTP protocol that Stryx extends. ARIA executes transactions in epochs: all transactions in an epoch are executed speculatively, conflicts are detected from read/write sets, and conflicting transactions are either reordered or deferred to the next epoch. The lock-free commit phase commits non-conflicting transactions, while a fallback lock-based phase handles the rest. Stryx extends ARIA with a function acknowledgment scheme for tracking transaction boundaries in arbitrary call graphs and a call graph cache that speeds up re-execution [28].

3.3.2. Other Deterministic Protocols

Starry [38] targets replicated databases with a semi-leader protocol for multi-master transaction processing. Lotus [39] focuses on multi-partition workloads using a run-to-completion single-thread (RCST) approach. They address replication and transaction scheduling rather than checkpoint persistence.

3.4. State Migration and Elasticity

Several streaming systems support dynamic reconfiguration and state migration [18, 15, 8, 20, 12]. Megaphone [18] interleaves small migration batches with normal processing to limit latency spikes. Meces [15] migrates frequently accessed keys first. Rhino [8] combines proactive incremental state replication with a handover protocol for large distributed state.

These systems scale proactively in response to load. We instead rebalance after an unexpected failure and reclaim partitions when a worker returns. Rather than transfer live state between workers, we reuse the existing recovery path and restore from a common checkpoint.

3.5. LSM-Tree Compaction

Log-Structured Merge trees (LSM trees) [27] are the dominant storage engine design behind RocksDB [10], LevelDB, and Apache Cassandra [22]. In an LSM tree, writes accumulate in an in-memory *memtable*, are flushed as immutable sorted runs (*SSTables*) organized into *levels*, and a background compaction merges adjacent levels to bound read amplification and reclaim space.

T-Statefun [16] inherits LSM-tree compaction from RocksDB. The original Stryx already merges immutable epoch-delta files to reduce recovery I/O (Section 2.4). We adapt this simpler scheme to uncoordinated mode by moving compaction from the coordinator to workers and adding configurable triggers (Section 4.3).

3.6. Summary and Research Gap

For Stryx, recovery can select the latest checkpoint epoch present for every partition. Compaction must preserve such an epoch while workers upload files asynchronously. Lockstep epoch boundaries avoid message logging and the domino effect. Chapter 4 develops this design, and the later chapters evaluate it.

Table 3.1: Positioning of this work relative to related systems. Among the systems compared here, this work is the only one that combines uncoordinated checkpointing, a recovery line computed from per-partition files, and worker-driven compaction on a streaming dataflow.

System	Checkpoint	Recovery line	Compaction
Beldi [36]	Per-function log	Per-workflow log replay	N/A
Boki [19]	Shared log	Shared-log position	N/A
T-Statefun [16]	Coordinated (Flink)	Latest complete checkpoint	LSM-tree
Flink [5]	Coordinated (barriers)	Latest complete checkpoint	N/A
Clonos [32]	Coordinated + causal log	Latest checkpoint + causal log	N/A
Calvin [35]	Decoupled (determinism + input replay)	Latest complete checkpoint	N/A
CALC [30]	Async (virtual points of consistency)	Latest complete checkpoint	N/A
Stryx (original)	Coordinated	Global file: min snapshot-id over workers	Epoch-delta [*]
This work	Uncoordinated	Newest epoch shared by all partition files	Epoch-delta[†]

^{*}Coordinator-driven compaction after each completed cluster snapshot.

[†]Worker-driven compaction with configurable trigger strategies and owned-partition scoping (see Section 4.3).

4

Design and Implementation

4.1. Uncoordinated Checkpointing with Epoch Alignment

4.1.1. Motivation

In coordinated checkpointing (Section 2.4), workers save their state and the coordinator writes one global sequencer file. A checkpoint is unusable until all of these writes finish. Uncoordinated checkpointing removes the coordinator write, but worker and storage failures can still stop checkpoint progress. It also leaves execution unchanged: both modes use the coordinator for every ARIA barrier. The expected benefit is therefore one fewer persistence dependency, not higher throughput. At the tested single-node scale, removing it has no measurable steady-state cost (Section 6.1).

Independent wall-clock timers can make workers checkpoint different epochs. Workers execute in lockstep (Invariant 2.3), but scheduling jitter, upload time, and small differences in epoch duration make their timers fire at different boundaries.

In *coordinated* mode, the coordinator absorbs this drift. It tracks $\min_{w \in W}(\text{snapshot_id}(w))$ and writes one file whose partition offsets may come from slightly different epochs. Deterministic replay bridges the gap (Section 2.4).

In *uncoordinated* mode, recovery must find an epoch present in every partition's files. A single shared epoch ensures that all files contain the same `t_counter` for restarting TID assignment (Invariant 2.2). Even one epoch of drift can make the intersection empty: partition *A* may have epoch 100 but not 101, while partition *B* has epoch 101 but not 100. Because each worker restarts its timer after checkpointing, this drift can grow over time:

Example 4.1 (Divergent Checkpoints with Pure Time-Based Triggering). *Consider four workers checkpointing every 10 seconds at a throughput where one epoch takes approximately 100 ms. Small timer variations of a few epochs per checkpoint compound over time:*

W_1 checkpoints at epochs: $\{100, 198, 301, 397, 502, \dots\}$

W_2 checkpoints at epochs: $\{100, 205, 308, 415, 519, \dots\}$

W_3 checkpoints at epochs: $\{100, 195, 294, 390, 488, \dots\}$

W_4 checkpoints at epochs: $\{100, 202, 299, 403, 510, \dots\}$

The intersection is $\{100\}$: only the first checkpoint is shared. A failure at epoch 500 requires rolling back to epoch 100, losing 400 epochs of work. In practice, the system would almost always recover from the very first checkpoint (or from epoch 0 if none is shared), making all subsequent checkpoints useless for recovery.

4.1.2. Epoch-Aligned Time-Based Strategy

The *epoch-aligned time-based* strategy limits this drift. Each worker has a local timer and an *alignment interval* A . It uses the following rule:

Algorithm 1: Epoch-Aligned Checkpoint Decision

Input: Current epoch e , time since last checkpoint Δt , time interval T , alignment interval A , snapshot-in-progress flag *busy*
Output: Boolean: should checkpoint

```

1 if  $e \leq 0 \vee (e \bmod A \neq 0)$  then
2   | return false;                                // Fast exit: not an aligned epoch
3 if busy then
4   | return false;                                // Previous snapshot still in progress
5 if  $\Delta t \geq T$  then
6   | return true;                                // Enough time has elapsed and epoch is aligned
7 return false;
```

Every worker evaluates the rule at the same epoch boundary (Invariant 2.3). Only multiples of A are candidates. Workers usually reach these boundaries at nearly the same wall-clock time, so their timers usually select the same candidate.

“Uncoordinated” applies only to checkpointing. Execution remains lockstep because every epoch still ends at the ARIA barriers (Invariant 2.3). Checkpointing neither adds nor removes these barriers. Without them, workers could be on different epochs and would have no shared checkpoint boundary.

A checkpoint has two steps. At the boundary, the worker synchronously copies its delta map. A background thread then uploads the copy while the next epoch runs. If that upload is still active at the next trigger, the snapshot-in-progress guard skips the trigger instead of blocking execution (Algorithm 1).

Example 4.2 (Aligned Checkpoints). With $A = 50$, $T = 10$ s, and epochs taking approximately 100 ms each:

- At epoch 50 (~5 s elapsed): all workers check, but $\Delta t \approx 5 \text{ s} < T$. No checkpoint.
- At epoch 100 (~10 s elapsed): all workers check, $\Delta t \approx 10 \text{ s} \geq T$. All checkpoint.
- At epoch 150 (~5 s since last checkpoint): $\Delta t < T$. No checkpoint.
- At epoch 200 (~10 s since last checkpoint): $\Delta t \geq T$. All checkpoint.

All workers checkpoint at $\{100, 200, 300, \dots\}$. The recovery line at epoch 270 is epoch 200.

Invariant 4.1 (Epoch Alignment). In uncoordinated mode with epoch-aligned time-based triggering, every checkpoint epoch is a multiple of A : for any worker w and checkpoint epoch $e_c \in \text{checkpoints}(w)$, $e_c \in \{kA \mid k \in \mathbb{N}_1\}$. Under normal conditions, all workers checkpoint at the same aligned epochs because they evaluate the same condition ($\Delta t \geq T$) at the same epoch boundaries. If a previous upload is still running at an aligned epoch, however, the snapshot-in-progress guard skips that trigger. Workers may therefore save different subsets of aligned epochs and select different later candidates.

The recovery line is always a multiple of A . Usually $A \times d < T$, where d is the average epoch duration, so several aligned candidates occur during each timer period. The timer selects every $\lceil T/(A \times d) \rceil$ -th candidate.

4.1.3. Trade-offs and Recovery Line Bounds

Because workers check the timer only at multiples of A , the interval is $\lceil T/(A \times d) \rceil \times A$ epochs, where d is the average epoch duration. With $A = 50$, $d \approx 100$ ms, and $T = 10$ s, this gives 100 epochs, or about 10 seconds.

If every scheduled trigger succeeds and each upload finishes within one interval, a failure loses at most one checkpoint interval of work. One additional interval may be lost if the latest upload is still active. Epoch alignment does not remove replay: a failure between checkpoints returns to the earlier checkpoint. Under the defaults, this is at most 100 epochs. The bound includes the checkpoint epoch and one epoch that may commit while failure detection is in progress (Section 4.4). Coordinated checkpointing has the same one-interval lag under these assumptions.

The bound falls to A epochs if every partition saves every aligned epoch, which requires $T \leq A \times d$. This condition is fragile. Different upload times can make one worker skip a candidate, especially when T is close to a multiple of $A \times d$. The missed epoch then disappears from the cross-partition intersection. Because the worker restarts its timer after the next successful checkpoint, one skip can also shift later checkpoints onto a different subset of aligned epochs. Operators should therefore keep T away from exact multiples of $A \times d$. With $d \approx 100$ ms, the defaults lie on such a boundary ($T = 2 \times A \times d$); setting $T = 12$ s would move it off the boundary.

The pure epoch-based trigger avoids this timer problem. Every worker targets $\{N, 2N, 3N, \dots\}$, giving an N -epoch bound, apart from an in-flight upload. If the busy guard skips one target, the worker rejoins the shared schedule at the next multiple of N . Wall-clock spacing varies with throughput, but faster processing also produces more state changes and warrants more checkpoints. We recommend the epoch-based trigger for a firm recovery bound. The aligned time-based trigger is useful when approximate wall-clock spacing matters more than occasional skips. In a lockstep system, the shared logical clock is the safer trigger.

The scalability experiments use epoch-based triggering and target the coordinated mode's 10-second cadence (Equation (5.1)). Actual spacing grows with worker count (Chapter 5). The checkpoint-strategy sweep therefore tests several epoch intervals and one aligned time-based configuration (Section 6.1.4).

4.2. Per-Partition Sequencer Files

4.2.1. Design

The global sequencer file is the remaining coordinator dependency. In uncoordinated mode, each worker instead writes a sequencer file for every owned partition with a non-empty delta. Our workloads modify every partition in every interval, so all owned partitions write files. Workers still send SnapID notifications for monitoring, but durability and checkpoint progress do not depend on them. The coordinator reads stable storage only during recovery to find the recovery epoch (Section 4.4.3).

4.2.2. Storage Format

The per-partition sequencer file for operator op , partition p , and epoch e is stored at:

`sequencer/{op}/{p}/{e}.bin`

and contains the tuple:

$$\text{SeqFile}(op, p, e) = (\text{input_offset}, \text{output_offset}, \text{epoch}, \text{t_counter}) \quad (4.1)$$

where:

- **input_offset**: the Kafka input topic offset for this partition at epoch e ,
- **output_offset**: the Kafka output topic offset for this partition at epoch e ,
- **epoch**: the epoch counter e ,
- **t_counter**: the global transaction counter, which is identical across all workers at epoch boundaries due to Invariant 2.2.

4.2.3. Write Order Invariant

When persisting a checkpoint, the sequencer file is always written *before* the corresponding data file:

Invariant 4.2 (Write Order). *For each checkpoint at epoch e for partition (op, p) :*

$$t_{\text{write}}(\text{sequencer}(op, p, e)) < t_{\text{write}}(\text{data}(op, p, e))$$

If a crash occurs between the two writes, it can leave a sequencer file without a data file, but not the reverse. We assume atomic PUTs and read-after-write consistency, as provided by our storage layer and Amazon S3 [2]. Recovery ignores orphan sequencer files because it computes the recovery epoch from data files (Equation (4.2)).

4.2.4. Recovery with Per-Partition Sequencer Files

During recovery, the coordinator lists each partition's data files and reads their epoch numbers from the filenames. The latest epoch present for every partition is the *recovery epoch*:

$$\text{recovery_epoch} = \max \left(\bigcap_{(op, p) \in \mathcal{P}} \text{epochs}(op, p) \right) \quad (4.2)$$

Here $\mathcal{P}$ contains the partitions with at least one data file, and $\text{epochs}(op, p)$ lists the stored epochs for a partition. A partition with no file restores as empty. This is safe only if it had no committed mutations; Proposition 4.1 therefore assumes that every partition that received input has checkpointed at least once.

Recovery intersects data files rather than sequencer files. The Write Order Invariant (Invariant 4.2) ensures that every data file has a matching sequencer file. Compaction preserves this ordering by processing data before sequencer files with the same cutoff E_{cut} . A crash between those passes may leave extra sequencer files, but cannot leave too few. One corner case remains: an older orphan sequencer file inside the compaction range can displace the file matching the compacted data epoch. Retaining the newest sequencer file at or below the newest data epoch would close this gap.

With epoch-based triggering (every N epochs), the intersection in Equation (4.2) contains only multiples of N ; with epoch-aligned time-based triggering (Invariant 4.1), it contains only multiples of A . In both cases, the recovery epoch is the most recent common checkpoint epoch across all partitions (Figure 4.1).

If the intersection is empty (no epoch has checkpoint files for every partition), no common recovery epoch exists, and compaction rules out falling back to an older one (a compacted file cannot restore any

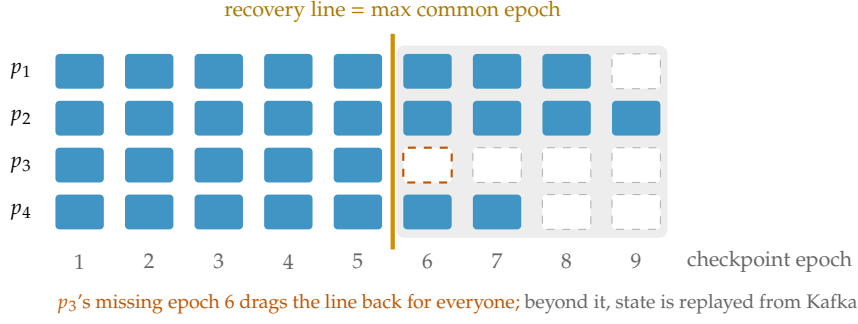


Figure 4.1: The recovery line of Equation (4.2). Filled cells are persisted epoch-delta files; each partition checkpoints independently. Recovery restores every partition to the newest epoch they *all* share (here epoch 5) and replays the remainder from Kafka, so one lagging partition bounds the line for the whole system.

epoch before its own; Section 4.3). The system then restarts from empty state (recovery epoch -1) and replays all transactions from the initial Kafka offsets.

Unlike the coordinated recovery line $\min_{w \in W}(\text{snapshot_id}(w))$ (Section 2.4), which operates at worker granularity (one snapshot identifier covering all of a worker's partitions), this formula operates at partition granularity: each partition has independent checkpoint files, so the recovery epoch must account for every partition individually.

4.2.5. Correctness Argument

The correctness of per-partition sequencer files relies on two properties:

1. **Partition ownership exclusivity** (Invariant 2.1): Each partition is owned by exactly one worker, so no two workers concurrently write sequencer files for the same partition.
2. **$t_counter$ consistency** (Invariant 2.2): The $t_counter$ stored in any partition's sequencer file is the same value across all partitions at the same epoch, because the AriaCommit barrier synchronizes it. This means recovery can use the $t_counter$ from any single sequencer file to initialize the global counter.

4.3. Epoch-Delta Compaction

4.3.1. Problem: Delta File Accumulation

Each checkpoint adds one delta file per modified partition. After n checkpoints, recovery must fetch and merge up to n files per partition. Recovery slows for two reasons:

1. **I/O overhead:** Each file requires a separate stable storage fetch operation.
2. **Merge overhead:** Files are deserialized and merged sequentially.

4.3.2. From Coordinator-Driven to Worker-Driven Compaction

In coordinated mode, the coordinator compacts after every cluster snapshot because it already tracks snapshot progress (Section 2.4). In uncoordinated mode, each worker tracks and compacts its own partitions. Compaction can then run less often to reduce normal-operation overhead.

4.3.3. Compaction Algorithm

Compaction merges eligible delta files for a given partition into a single consolidated file (Figure 4.2):

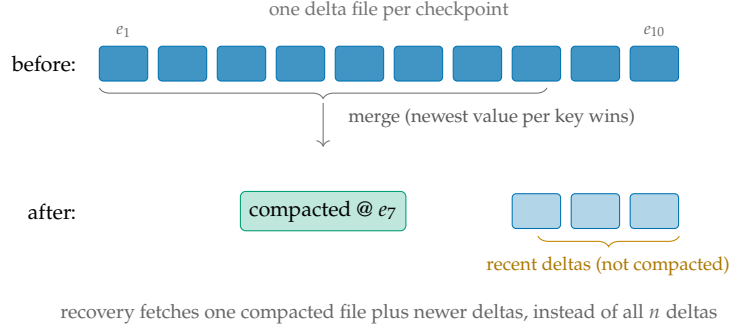


Figure 4.2: Epoch-delta compaction for one partition. Deltas through the compaction cutoff are merged into one file (newest value per key wins); newer deltas remain unchanged. The cutoff stays at least B epochs behind the current epoch so that a concurrent rebalance can still find a common recovery epoch (Section 4.5.4).

Algorithm 2: Epoch-Delta Compaction

Input: Stable storage client, compaction cutoff E_{cut} , owned partitions $\mathcal{P}_w$

```

1 foreach  $(op, p) \in \mathcal{P}_w$  do
2   files  $\leftarrow$  list files under  $data/\{op\}/\{p\}/*.bin$ ;
3   eligible  $\leftarrow \{(e, f) \in \text{files} \mid e \leq E_{cut}\}$ ;           // newer files remain unchanged
4   if  $|eligible| < 2$  then
5     continue;
6    $e^* \leftarrow \max\{e \mid (e, f) \in eligible\}$ ;
7   merged  $\leftarrow \{\}$ ;
8   foreach  $(e, f)$  in  $sorted(eligible)$  do
9     delta  $\leftarrow deserialize(f)$ ;
10    merged.update(delta);
11  write( $data/\{op\}/\{p\}/\{e^*\}.bin$ , merged);           // overwrites existing delta
12  delete files where  $e < e^*$ ;

```

Files are merged in epoch order, so the newest included value for each key wins. The result is written as $e^*.bin$, where e^* is the newest stored checkpoint at or before E_{cut} . This write replaces the original delta at e^* , and files from earlier epochs are then deleted. Files newer than E_{cut} remain ordinary, uncompactd delta files. Because the filename records the newest epoch represented in the merged state, the recovery-line calculation remains valid (Equation (4.2)).

The worker leaves recent epochs uncompactd by setting

$$E_{cut} = e_{\text{current}} - \max(C + 1, B).$$

The $C + 1$ term keeps compaction behind the current checkpoint and any write still in progress. The $B = 100$ term keeps the cutoff far enough behind the current epoch for rebalancing (Section 4.5.4). Thus, B specifies a minimum cutoff margin in epochs, not a fixed number of retained files. The trigger counts checkpoint events, but the cutoff is expressed in epochs. These units are not identical because an asynchronous upload can prevent a checkpoint in the next epoch. In the compaction experiments, roughly two epochs produced one stored checkpoint.

4.3.4. Scoped Compaction

Each worker compacts only the partitions it owns. Otherwise, two workers could read the same delta files and race to replace and delete them.

The `owned_partitions` parameter enforces one compactor per partition. Data and sequencer compaction also list separate prefixes, so neither can delete the other's files.

Mid-compaction crash risk. If a worker crashes after writing the compacted e^* .bin file but before deleting the old delta files, the partition has both the compacted file and the original deltas. This is safe: recovery loads all available files and merges them in epoch order, and re-applying deltas already included in the compacted file is idempotent (dictionary update semantics). If the crash occurs during the compacted file write itself, the atomic PUT assumption ensures the partial file is not visible, and recovery falls back to the remaining delta files. The worst case is temporary space amplification, not data loss.

4.3.5. Sequencer File Compaction

Per-partition sequencer files are compacted in the same way. The latest sequencer file with epoch at or below E_{cut} is retained under its own epoch name, and older files for that partition are deleted. As with data files, the filename preserves the actual epoch, keeping the recovery line computation correct.

4.3.6. Interaction with Recovery Line Computation

Each worker compacts its partitions in one pass. Before compaction, those partitions normally share the same checkpoint epochs; this holds in our workloads because every partition changes during every interval. A long-idle partition would instead hold back the recovery line. The loop handles partitions one at a time, writing the merged file before deleting old deltas. Four cases matter:

During file deletion. After writing e^* .bin but while old files are being deleted, the partition's epoch set contains e^* and any not-yet-deleted delta epochs. This cannot remove an epoch from the recovery-line intersection.

Listing-fetch window. A restore reads a partition in two steps: it lists the partition's files, then fetches them. A compaction that replaces old deltas between a restore's listing and its fetches would make the listed files unfetchable, even though their contents live on in the new merged file. We assume the two do not interleave within this window. Every recovery with a live workload passed consistency verification, so none exposed this race. A production implementation should re-list and retry after a failed fetch rather than continue without the file.

Between partitions. If a worker crashes after compacting partition P_1 to epoch 100 but before compacting partition P_2 (which still has deltas at {50, 100, 150}), the intersection includes epoch 100 (present in both P_1 's compacted file and P_2 's delta). Recovery at epoch 100 loads P_1 's compacted **100**.bin (state through epoch 100) and P_2 's deltas up to epoch 100, yielding a consistent global state. Naming the compacted file after its actual checkpoint epoch therefore avoids a sentinel fallback to full Kafka replay.

Across workers. Since compaction is uncoordinated, different workers may compact at different epochs; the between-partitions argument applies unchanged: the intersection retains any epoch present in every partition's file set, whether as a compacted file or a delta. A freshly compacted partition whose only file is `100.bin` still contributes `{100}`.

4.3.7. Compaction Trigger Strategies

We implement and evaluate three compaction trigger strategies:

No compaction ($C = 0$): Delta files accumulate without bound. Used as baseline.

Periodic compaction (interval C): Compact every C checkpoints; tested with $C \in \{1, 10, 100, 1,000\}$ (the load sweep of Chapter 5 adds $C = 50$).

Recovery-only compaction: No compaction during normal operation. Upon recovery, all accumulated delta files are merged by a background thread after state restoration, so the first restoration follows the baseline code path and later recoveries benefit only once that merge completes (Section 6.2.6).

4.4. Failure Detection and Recovery

4.4.1. Failure Detection

STYX uses two failure detection mechanisms:

Instant restart detection (~100 ms): When a worker process crashes and restarts at the same network address, it re-registers with the coordinator. The coordinator detects a restart when two conditions hold simultaneously: (1) the worker's network address is already mapped to a registered worker ID, and (2) the registration message has `reconnecting=False`, distinguishing a fresh start from a network reconnection. When both conditions are met, the coordinator immediately triggers recovery.

Heartbeat timeout (~5 s): Each worker sends periodic heartbeats (interval: 500 ms). If no heartbeat is received within the heartbeat limit (default: 5 seconds, configurable), the coordinator suspects the worker and, after a 3-second confirmation window, marks it as failed (in the Chapter 6 timeline estimates, this confirmation window is folded into the redistribution wait that follows detection). This handles network partitions and worker hangs where no restart occurs.

The five-second detection delay is not inherent. Because the coordinator runs every barrier, it could detect a missing worker within about one epoch (~100 ms) and use heartbeats only while idle. We retain the conservative heartbeat timeout. Together with restart, it accounts for most of the ~14 s unavailability floor (Section 6.2.7).

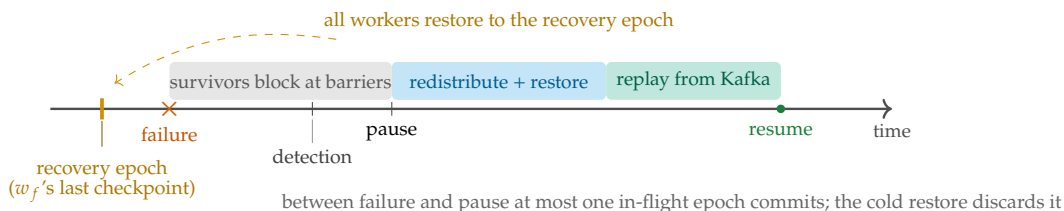


Figure 4.3: Failure-to-resume sequence in uncoordinated mode. Survivors stall at the first barrier missing the failed worker until the pause releases them; every worker then restores to the recovery epoch, which predates the failure, and deterministic replay with output deduplication re-executes the gap.

4.4.2. Recovery Process

When a worker failure is detected, recovery proceeds as follows (Figure 4.3 summarizes the sequence):

1. **Pause survivors.** The coordinator sends `PauseForRecovery` to all surviving workers. Each surviving worker blocks after aborting its in-progress epoch at the next barrier, preventing epoch divergence.
2. **Partition redistribution.** The coordinator redistributes the failed worker's partitions among survivors using round-robin assignment, ensuring all partitions remain served.
3. **Compute recovery epoch.** The coordinator finds the latest epoch stored for every partition (Equation (4.2)). Storage is authoritative because in-memory metadata may claim an epoch whose writes did not all succeed (Figure 4.4).
4. **Global recovery.** The coordinator sends `InitRecovery` to every worker. All workers restore their assigned partitions to the same epoch, including workers whose assignments did not change.
5. **Resume processing.** `InitRecovery` includes the new routing table. After every worker reports `ReadyAfterRecovery`, the coordinator updates barrier membership and resumes processing. Kafka replay re-executes work between the recovery epoch and the pause.

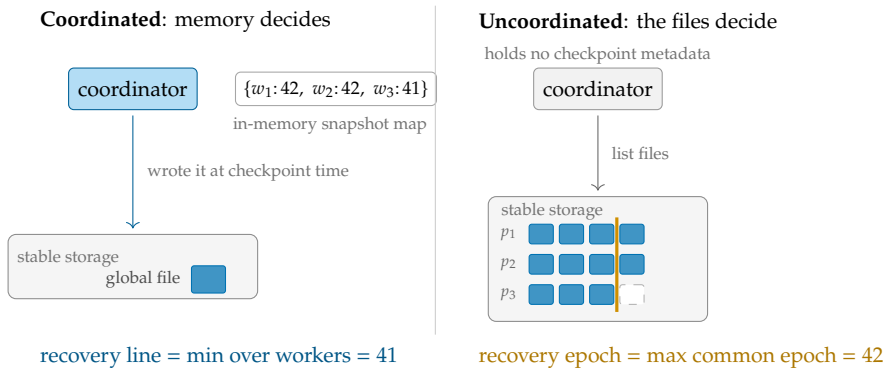


Figure 4.4: Where the recovery point comes from. Coordinated recovery trusts the coordinator's in-memory snapshot map and takes the minimum over workers. Uncoordinated recovery trusts only stable storage: the coordinator lists the per-partition files and takes the newest epoch they all share, so exactly what was persisted, and nothing else, determines the recovery point.

4.4.3. Recovery Epoch Computation from Stable Storage

In both redistribution recovery and rebalance recovery, the coordinator computes the recovery epoch by querying stable storage directly. It lists the data files under the `data/` prefix and applies Equation (4.2).

Invariant 4.3 (Recovery Epoch Source of Truth). *In uncoordinated mode, the recovery epoch is always computed from stable storage contents as $\max\left(\bigcap_{(op,p) \in \mathcal{P}} \text{epochs}(op, p)\right)$, not from worker-local or coordinator-local metadata.*

4.4.4. In-Progress Epochs During Recovery

Recovery usually discards the in-progress epoch. An epoch has no committed effects before `AriaCommit` (Section 2.3), so it can run again after recovery. At most one epoch can still commit after the failure: an epoch for which the failed worker had already sent its `AriaCommit` contribution. Those commits remain only in survivor memory. Restoring every worker to an epoch shared by all partitions removes them and prevents divergent state.

4.4.5. End-to-End Recovery Correctness

SRYX’s original exactly-once argument covers the coordinated recovery line and global sequencer file (Lemma 7.1 in [28]). The following extends it to per-partition files and compaction. First, compaction must preserve this property:

Lemma 4.1 (Delta Prefix Completeness). *For every partition (op, p) and every epoch $e \in \text{epochs}(op, p)$, the union of the partition’s retained data files with epoch $\leq e$ contains every key the partition mutated in epochs $[0, e]$, and merging those files in ascending filename-epoch order yields the newest committed value per key: the partition’s state at the boundary of epoch e .*

Proof sketch. Before any compaction, the delta files partition the mutation history by checkpoint interval, so the property holds by construction. Compaction (Algorithm 2) replaces the files with epoch $\leq E_{cut}$ by one file, named after the largest included epoch, containing their merged contents; coverage and merge order are preserved, and files above E_{cut} are untouched. Compaction reads every file it merges; under the system model’s storage availability assumption a read does not fail, and the implementation additionally abandons a partition’s merge on a failed read rather than write a merged file whose name overstates its coverage. A mid-compaction crash leaves a superset of a correct file set, and re-merging duplicated coverage is idempotent under dictionary-update semantics (Section 4.3.4). $\square$

Proposition 4.1 (Recovery Correctness). *Assume the system model of Section 2.2, the listing–fetch condition of Section 4.3.6, and the output condition of Section 2.5. Also assume that every partition receiving input has a data file at some epoch $\leq R$, as in our workloads. If the checkpoint intersection is non-empty with maximum R , recovery restores the committed global state at epoch R . Replay then reproduces every later transaction and delivers each output exactly once.*

Proof sketch. *State.* R lies in every partition’s epoch set, so by Lemma 4.1 merging each partition’s data files with epoch $\leq R$ yields that partition’s state at the boundary of R . *Consistency.* That boundary is a consistent cut (Proposition 2.1), and every partition’s sequencer file at R records the same $t_counter$ (Invariant 2.2) together with its own Kafka offsets. The assembled state, counter, and offsets therefore describe one global state that actually occurred. *Replay.* Deterministic execution (Section 2.5, property 1) from that state over the input replayed from the recorded input offsets (property 2) reproduces every post- R transaction’s outcome. Reading the output topic forward from the recorded output offsets identifies outputs already delivered, whose re-emission is suppressed (property 4), so each output is delivered exactly once. $\square$

The proposition does not cover an empty intersection; that case restarts from empty state and replays the full input (Section 4.2). It also requires Kafka to retain the entire replay range (Section 2.2).

4.5. Partition Rebalancing

4.5.1. Motivation

After a failure and recovery, the failed worker’s partitions have been redistributed among survivors. When the failed worker eventually restarts and rejoins the cluster, the partition distribution is imbalanced: some workers host more partitions than others, reducing overall throughput.

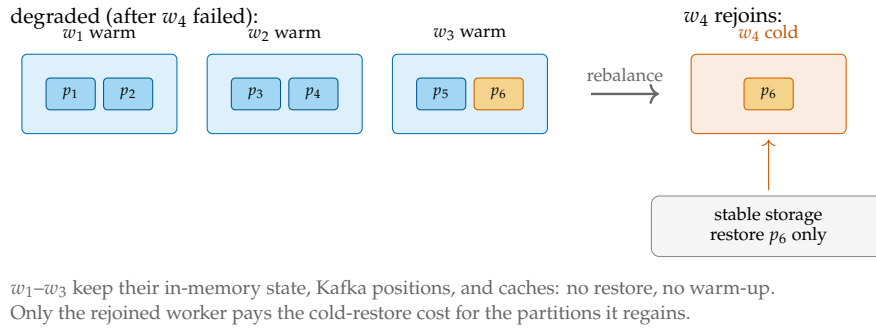


Figure 4.5: Selective restore during a Phase 2 rebalance ($w = 4$, one failure, minimal redistribution). The rejoined worker cold-restores only the partition it regains (p_6); the three warm workers are untouched. Contrast with Phase 1, where safety requires every worker to cold-restore (Section 4.4).

4.5.2. Rebalance Recovery Protocol

Recovery has two phases. Immediate redistribution uses a *full cold restore*: every worker reloads the checkpoint. A later rebalance uses *selective restore*: only workers that gain partitions reload.

Phase 1: Redistribution recovery (full cold restore). Survivors may contain commits that the failed worker never checkpointed, including at most one epoch committed after the failure (Section 4.4). Every survivor must therefore roll back to the shared checkpoint. Selective restore in this phase caused sporadic consistency violations in testing (Section 7.3).

Phase 2: Rebalance (selective restore, Figure 4.5). When the worker returns, the coordinator pauses everyone at the same completed epoch. Workers that gain partitions restore them from storage; all others keep their in-memory state and receive only a routing update.

Warm vs. cold workers. A *warm* worker keeps its memory state, Kafka position, and runtime caches. A *cold* worker has just loaded a checkpoint. Its state is correct, but it initially processes epochs more slowly (Section 6.3.2). At high load, this difference determines whether the cluster can drain the backlog created during the pause.

1. **Pause all workers.** The coordinator sends `PauseForRecovery` to all existing workers (including the rejoined worker). Workers block after finishing or aborting their in-progress epoch at the next barrier and report their paused epoch to the coordinator.
2. **Minimal redistribution.** The coordinator preserves existing assignments and moves only excess partitions from overloaded to underloaded workers. With 4 workers and 1 failure, for example, W_3 may inherit P_3 from W_4 . When W_4 returns, only P_3 moves back; every original assignment stays unchanged. This keeps as many workers warm as possible.
3. **Classify workers.** The coordinator compares each worker's old and new assignments and classifies them into three categories:
 - *Unchanged*: identical assignments before and after (e.g., $W_1: [P_0] \rightarrow [P_0]$). These workers keep their warm in-memory state. With minimal redistribution, this is the common case: most workers are unchanged.

- *Shrunk*: lost partitions but retained others (e.g., $W_3: [P_2, P_3] \rightarrow [P_2]$). These workers keep state for the retained partitions and drop the removed ones. An *ingress guard* defensively skips messages arriving for removed partitions (see below).
- *Gained*: acquired new partitions not previously owned (e.g., $W_4: \emptyset \rightarrow [P_3]$). These workers must restore from stable storage (cold).

For a typical $w = 4$, $f = 1$ rebalance with minimal redistribution, this yields 2 unchanged + 1 shrunk = 3 warm workers and 1 cold (gained) worker, so only 1 of 4 workers pays the cold-restore cost.

4. **Epoch alignment.** The coordinator queries stable storage for the latest common checkpoint epoch. If the stable storage epoch is behind the paused epoch (due to asynchronous checkpoint uploads), the coordinator polls briefly. If alignment fails, it falls back to full `InitRecovery` for all workers to guarantee epoch consistency (Invariant 2.3).
5. **Selective recovery.** The coordinator sends `InitRecovery` (full cold restore) to gained workers and `ReceiveExecutionPlan` (lightweight routing update) to unchanged and shrunk workers. Both message types cause the worker to signal `ReadyAfterRecovery`.
6. **Sequencer buffer reset.** Warm workers perform a three-step reset to align with cold workers (see Section 4.5.2).
7. **Resume processing.** After all workers signal ready, the coordinator broadcasts a cluster-healthy notification. Unchanged and shrunk workers resume their paused protocol with cleared transient state; gained workers start a fresh protocol from the checkpoint.

With f failed workers, minimal redistribution creates f cold workers and can keep up to $w - f$ workers warm, depending on the partition distribution.

Sequencer Buffer Reset

During the rebalance pause, the ingress task continues consuming from Kafka and buffering messages in the sequencer's distributed log, advancing the transaction counter (`t_counter`). Cold workers restore `t_counter` from the checkpoint (a lower value). To prevent state divergence, warm workers perform a three-step reset:

1. *Reset `t_counter`*: subtract the buffer length to reverse the advancement caused by ingress buffering, aligning with the cold worker's checkpoint value.
2. *Clear the buffer*: discard messages that reference pre-rebalance partition assignments.
3. *Seek the Kafka consumer back*: reset the consumer to the last committed offset so the cleared messages are re-delivered and re-sequenced with the corrected `t_counter`.

Omitting any step causes warm-cold state divergence. In the original design, the sequencer modulus M of the TID scheme (the n_{seq} of Equation (2.1)) equals the number of active workers (n_{workers}), which changes during failure and recovery ($4 \rightarrow 3 \rightarrow 4$). Changing M invalidates any transaction IDs buffered during the pause, because IDs generated with the old M may collide with IDs generated by other workers under the new M . For example, worker 1 with $M = 4$ generates $\text{TID} = 1 + 1 \times 4 = 5$; worker 2 with $M = 3$ generates $\text{TID} = 2 + 1 \times 3 = 5$. Both would emit TID 5, a collision.

We replace M with $n_{\text{partitions}}$, which is fixed at graph submission time and never changes. With $n_{\text{partitions}} = 4$ and 3 active workers, each worker's transaction IDs occupy a distinct residue class modulo 4, leaving one class unused. A fixed modulus remains unique only while the number of active workers does not exceed the partition count. This condition holds in our deployments: the partition count equals the initial worker count, and rejoining workers never raise the active count above it. When the fourth worker rejoins, it takes the unused class. Transaction IDs therefore remain valid as the worker count changes, eliminating the need for re-sequencing.

4.5.3. Correctness

The protocol pauses all workers at an epoch boundary, changes partition ownership, and resumes everyone at one epoch. Correctness depends on four properties:

- **Epoch alignment** (Invariant 2.3): warm workers remain at the paused epoch; cold workers restore to the same epoch from stable storage. The recovery epoch is determined by the *intersection* of epoch sets across all partitions: only epochs for which every partition has an actual checkpoint file are valid, because a compacted file at epoch N contains cumulative state through N and cannot be used to restore to any earlier epoch (the individual deltas were deleted). If the intersection is behind the paused epoch, the fallback restores all workers from stable storage; if it is empty, recovery restarts from empty state (epoch -1) with full input replay.
- **Transaction counter alignment** (Invariant 2.2): warm workers' `t_counter` may differ from cold workers' (checkpoint value) due to ingress buffering during the pause. The three-step sequencer reset (step 6) reverses this advancement. Even if a residual offset exists in the first epoch, the `AriaCommit` barrier synchronizes `t_counter` from the second epoch onward. Transaction IDs remain unique because each worker uses a distinct sequencer ID modulo $n_{\text{partitions}}$. Resetting `t_counter` too far back must be avoided, as it would reuse IDs from prior epochs.
- **State completeness**: unchanged workers already have complete state for their partitions. Shrunk workers drop only removed partitions and retain correct state for kept ones. Gained workers restore all state from the checkpoint.
- **Partition ownership** (Invariant 2.1): each partition is owned by exactly one worker after redistribution. The ingress guard on shrunk workers skips messages from removed partitions; only the new owner processes them.

Ingress guard. After a rebalance, a shrunk worker's Kafka consumer retains its original partition assignment and continues delivering messages for partitions that the worker no longer owns. Without a guard, these messages would be ingested and sequenced, leading to duplicate processing (the new owner also processes them). The ingress guard is a defensive check at the ingress path: before enqueueing a message, the worker looks up the message's partition in its current `owned_partitions` set. Messages for partitions not in the set are silently dropped. This is an $O(1)$ dictionary lookup per message and adds negligible overhead.

Limitation. The protocol's Kafka consumer is bound to its partition assignment at creation time and cannot be reassigned without a full protocol restart. For shrunk workers, this means the consumer continues fetching messages from removed partitions (which are discarded by the ingress guard). This

wastes a small amount of Kafka bandwidth proportional to the number of removed partitions. A full solution would require live Kafka consumer reassignment, which we leave to future work (Section 7.4.2).

4.5.4. Compaction–Rebalance Interaction

Between redistribution and rebalance, survivors may advance faster than the restarting worker. The active survivors remain in lockstep, but the restarting worker has not yet rejoined their barriers. Independent compaction can then remove their last shared epoch. For example, a warm partition may retain {208, 209, 210} while a cold partition has only {203, 204, 205, 206}. Their intersection is empty. Rebalance must then restore from empty state and replay the full Kafka input (Section 4.2).

To preserve a shared epoch, compaction leaves files newer than $E - B$ uncompactd. If workers diverge at rate r for Δt seconds, the cutoff margin needs $B \geq r\Delta t$. At 18,000 TPS, 4 workers, and epoch size 1,000, $r \approx 4.5$ –6 epochs/s. The measured ~9-second window therefore needs a margin of up to about 54 epochs. We use $B = 100$, giving roughly a 2× margin. Because checkpoints may skip epochs, this does not imply that exactly 100 delta files remain.

The empty-intersection fallback remains correct but requires full replay. A stronger design would always retain the newest epoch that the coordinator has observed for every partition, perhaps reported through heartbeats. That rule would preserve one shared epoch by construction. During normal operation all workers advance together, so the cutoff margin does not affect the intersection.

4.6. Differences Between Coordinated and Uncoordinated Modes

The transaction processing pipeline (the full ARIA barrier sequence, conflict detection, and commit logic) is *identical* in both modes during failure-free operation; after a detected failure, survivors block at the next barrier until the pause releases them, and the coordinator resets barrier membership to the surviving workers when it finalizes recovery (Section 4.4).

Table 4.1: Comparison of coordinated and uncoordinated checkpointing in SRYX. The transaction pipeline is identical in both modes; the two differ only in when checkpoints are triggered, who writes checkpoint state, and how the recovery line is computed.

Aspect	Coordinated	Uncoordinated
ARIA barriers per epoch	5 (3 mandatory + fallback pair)	5 (3 mandatory + fallback pair)
t_counter synchronization	AriaCommit	AriaCommit
Epoch execution	Lockstep	Lockstep
Checkpoint trigger	Time-based, arbitrary epochs	Epoch-based or epoch-aligned time-based*
Sequencer files	Global: sequencer/{snapshot_id}.bin	Per-partition: sequencer/{op}/{p}/{e}.bin
Sequencer writer	Coordinator	Workers
Recovery line	$\min_{w \in W}(\text{snapshot_id}(w))$	$\max\left(\bigcap_{(op,p)} \text{epochs}(op,p)\right)$
Compaction driver	Coordinator (after each snapshot)	Workers (configurable interval, scoped)
Coordinator role in checkpointing	Active (track, aggregate, write)	Monitoring only (fire-and-forget SnapID); authoritative at recovery

*Pure time_based triggering is also supported but not recommended due to recovery line degradation (Example 4.1).

The design relies on a property ARIA already provides: each epoch barrier creates a consistent cut. Workers can therefore decide locally whether to save that cut. See Appendix B.

5

Experiment Design

5.1. Methodology

5.1.1. General Principles

All experiments:

- **Repetitions.** Each configuration is run up to 5 times. We report the *median* to reduce the influence of outliers, with [min–max] ranges where dispersion matters, following standard practice in systems research (Section 5.6.2).
- **Warmup.** We collect no metrics during the first 10–30 seconds, allowing connections, Kafka consumer groups, and buffers to stabilize. The scalability suite and compaction Phases 1–2 use 10 s; compaction Phase 3 and the rebalance suite use 30 s.
- **Controlled environment.** All experiments run on the same hardware with identical Docker container configurations. External systems are deployed with fault-tolerant replication (Kafka: 3 brokers, MinIO: 4 nodes).
- **Consistency verification.** After every recovery experiment with a live workload, we verify transactional consistency: exactly-once output, zero missed messages, and zero wrong values.

5.1.2. Hardware and Software Environment

All experiments run on a single server with the configuration listed in Table 5.1.

5.1.3. Workload

All experiments use YCSB+T [9], adapted for Stryx. Each transaction simulates a bank transfer with two reads and two writes. Debtor and creditor accounts are selected from 10,000 keys using one of two distributions:

- **Uniform** (Zipfian constant = 0.0): Keys are selected uniformly at random. This represents low contention.

Table 5.1: Experiment server specification.

Component	Specification
CPU	AMD EPYC 7H12 (64 cores, 128 hardware threads)
RAM	512 GB DDR4
Storage	446.6 GB SSD (Dell PERC H755)
OS	Ubuntu Linux 5.15.0-133-generic (x86_64)
Docker	28.1.1
Python	3.10.12
Kafka	Confluent Platform 7.8.0
MinIO	RELEASE.2025-01-20T14-49-07Z (4 instances)

- **Skewed** (Zipfian constant > 0): The creditor key follows a Zipfian distribution, concentrating requests on a subset of keys. Higher values increase contention.

Using 10,000 keys creates moderate contention even under uniform access and exercises conflict detection.

Each key stores a single integer value (initial balance: 1,000,000), so total state is approximately 80 KB for 10,000 keys. This small state size means state restoration time is dominated by per-object I/O overhead rather than data volume.

5.1.4. Metrics

Throughput (TPS). Committed transactions per second, measured at the client.

Latency. End-to-end latency from client submission to response receipt. We report the 50th (P50), 95th (P95), and 99th (P99) percentiles. Output timestamps are producer-side commit timestamps.

State restoration time (ms). Time spent listing, fetching, and merging checkpoint files. It excludes failure detection, container restart, and Kafka replay; the total-unavailability model treats those separately. We divide restoration into:

- *List files*: Time to enumerate checkpoint files in stable storage.
- *Fetch files*: Time to retrieve delta and sequencer files from stable storage.
- *Merge data*: Time to deserialize and merge delta files.

When multiple workers restore simultaneously, we report the maximum across workers (the recovery bottleneck).

Delta file count. Number of accumulated delta files at the time of failure, directly affecting recovery I/O.

Consistency. Binary pass/fail: $\text{exactly_once_output} \wedge \text{missed_messages} = 0 \wedge \text{wrong_values} = 0$.

5.2. Cluster Configuration

Table 5.2 summarizes the cluster configuration used across all experiments unless otherwise noted.

Table 5.2: Default cluster configuration.

Parameter	Value
Workers	4 (varies in scalability experiments)
Partitions per operator	Equal to number of workers
Worker threads per worker	1 (single-threaded workers)
Kafka replicas	3
MinIO nodes	4
Client threads	2
Epoch size	1,000 transactions
Heartbeat send interval	500 ms
Heartbeat check interval	1,000 ms
Heartbeat timeout	5 seconds

5.3. Experiment Suite 1: Scalability (RQ1a–RQ1d)

This suite compares the steady-state performance of both checkpointing modes across varying worker counts, epoch sizes, and checkpoint strategies. It consists of four phases. Uncoordinated runs in this suite disable compaction ($C = 0$); coordinated mode compacts after each completed snapshot as part of its baseline behavior (Section 2.4).

5.3.1. Phase 1: Saturation Point Discovery (RQ1a, RQ1c)

Method. We binary-search total input rates from 2,000 to 60,000 TPS in 1,000 TPS steps. Each candidate runs for 30 seconds and is repeated 5 times. The *saturation point* is the highest rate whose median P95 stays below 1,000 ms. It describes this 30-second window, not indefinite operation. After testing the first mode, we search the second within $\pm 25\%$ of that result. A run fails after approximately 735 seconds.

Variables.

- **Independent:** Worker count $\in \{2, 4, 6, 8, 10, 12\}$; checkpointing mode: coordinated or uncoordinated.
- **Dependent:** Saturation TPS, P95 and P99 latency at saturation.
- **Fixed:** Epoch size = 1,000, Zipfian constant = 0.0.

Checkpoint strategy. Coordinated mode uses `time_based` triggering with a 10-second interval (the system default). Uncoordinated mode uses `epoch_based` triggering, checkpointing after a fixed number of epochs rather than at time intervals. The interval is computed from the coordinated saturation TPS to target approximately 10 seconds between checkpoints:

$$\text{interval} = \max\left(1, \text{round}\left(\frac{\text{TPS}_{\text{sat}}}{\text{epoch_size}} \times 10\right)\right) \quad (5.1)$$

Here TPS_{sat} is coordinated saturation TPS per client thread. Coordinated mode must therefore run first. The resulting intervals are 55, 95, 125, 160, 190, and 220 epochs for $w = 2$ through $w = 12$. Wall-clock

spacing varies with the achieved epoch rate. Phase 3 separately measures sensitivity to checkpoint frequency.

5.3.2. Phase 2: Epoch Size Impact (RQ1b)

Method. At the saturation TPS from Phase 1, we run the workload with varying epoch sizes, repeating each configuration 5 times. For uncoordinated configurations, the `epoch_based` checkpoint interval is recomputed for each epoch size to maintain approximately 10 seconds between checkpoints, so that the epoch size change does not confound checkpoint frequency.

Variables.

- **Independent:** Epoch size $\in \{1, 5, 50, 100, 250, 500, 2,000\}$ (plus 1,000 reused from Phase 1).
- **Dependent:** P95, P99 latency; throughput.
- **Fixed:** TPS from Phase 1, checkpoint frequency ≈ 10 s, all Phase 1 (worker, mode) configurations.

5.3.3. Phase 3: Checkpoint Strategy Sweep (RQ1c)

Method. At the saturation TPS from Phase 1, test checkpoint strategies with varying intervals. The Phase 1 baseline (`epoch_based` with the computed ~ 10 s interval) is included as a comparison point.

Variables.

- **Independent:** Checkpoint strategy (9 total):
 - `epoch_N` for $N \in \{1, 5, 10, 25, 50, 100, 250, 500\}$: checkpoint every N epochs.
 - `time_10s`: uses the `epoch_aligned_time_based` trigger with alignment interval $A = 50$ epochs and timer $T = 10$ s, providing a time-based comparison.
- **Dependent:** P95 latency, throughput ratio vs. baseline.
- **Fixed:** Mode = uncoordinated, TPS from Phase 1.

5.3.4. Phase 4: Contention Sweep (RQ1d)

Validation window. After each 30-second run, we wait up to 120 seconds for outputs before reading final state. Expected balances include every submitted transfer. If transactions remain queued, observed state will not yet match that expectation. Such a mismatch shows that the run did not reach quiescence; it does not by itself prove a serialization failure.

Variables.

- **Independent:** Zipfian constant $\in \{0.25, 0.5, 0.75, 0.99\}$ (the $z = 0.0$ uniform baseline is reused from Phase 1).
- **Dependent:** P95, P99 latency; throughput ratio vs. uniform baseline.
- **Fixed:** TPS from Phase 1, all Phase 1 (worker, mode) configurations.

5.3.5. Experiment Matrix Summary

Table 5.3: Scalability experiment matrix.

Phase	Variable	Values	Est. Experiments
P1: Saturation	Workers $\times$ Mode	$\{2, 4, 6, 8, 10, 12\} \times \{\text{Coord.}, \text{Uncoord.}\}$	$\sim 12 \times 7 \times 5 = 420$
P2: Epoch size	Epoch size	$\{1, 5, 50, 100, 250, 500, 2,000\}$	$12 \times 7 \times 5 = 420^\ddagger$
P3: Ckpt. strategy	Strategy (Uncoord. only)	9 strategies (8 epoch + 1 time)	$6 \times 9 \times 5 = 270$
P4: Contention	Zipf constant	$\{0.0^*, 0.25, 0.5, 0.75, 0.99\}$	$12 \times 4 \times 5 = 240$
Total (approximate)			$\sim 1,350$

* $z = 0.0$ reused from Phase 1 (not re-run in Phase 4). $z = 0.99$ experiments at $w = 10$ and $w = 12$ timed out: extreme contention with many workers reduced throughput so severely that the measurement window expired before sufficient outputs were collected (e.g., 29% completion at $w = 10$). This reduces the Phase 4 count by 20 runs.

‡ Epoch size 1 exceeded the experiment timeout at every worker count; the 60 affected Phase 2 runs produced no data (Section 6.1).

5.4. Experiment Suite 2: Compaction and Recovery (RQ2a–RQ2d)

This suite measures how epoch-delta compaction affects recovery. Phases 1 and 2 use epoch size 100 to accumulate more delta files within the measurement window; Phase 3 uses the default 1,000. All runs use uncoordinated mode, an epoch-based trigger, and uniform key access ($z = 0$).

5.4.1. Phase 1: Steady-State Overhead

Method. Run the workload at 1,000 TPS per client thread (2,000 TPS total) for 3,000 epochs under each compaction configuration with no failures injected.

Configurations.

1. **Baseline:** No compaction ($C = 0$, no recovery-only compaction).
2. **Continuous:** Compaction every $C \in \{1, 10, 100, 1,000\}$ checkpoints. (This suite triggers a checkpoint every epoch; in practice the busy guard leaves about one persisted checkpoint per two epochs, so C checkpoints span roughly $2C$ epochs; Section 6.2.)
3. **Recovery-only:** No compaction during operation; compact at recovery time.

Dependent variables. Throughput, P95 latency.

5.4.2. Phase 2: Recovery Performance

Method. After accumulating 3,000 epochs of checkpoints, kill 1 of 4 workers. Workers are terminated using `docker kill (SIGKILL)`, simulating an immediate crash with no opportunity for graceful shutdown. Measure state restoration time and its breakdown, validate consistency, and run a 60-second post-recovery phase. After redistribution completes, the killed worker is restarted immediately and rejoins, triggering a second recovery, the rebalance recovery (R2); both recoveries are timed. In this suite the rebalance recovery restores state on *all* workers; the selective-restore optimization evaluated in Suite 3 (Section 5.5) was introduced with the rebalance suite.

Variables.

- **Independent:** Compaction configuration (6 levels), failure count = 1.
- **Dependent:** State restoration time, restoration-time breakdown, delta file count, consistency.
- **Fixed:** Workers = 4, TPS = 1,000 per client thread (2,000 total), checkpoint every epoch, epoch size 100.

5.4.3. Phase 3: Load-Dependent Compaction Overhead

Scope. Phases 1 and 2 use only ~11% of capacity. Phase 3 tests 50–100% load to find where each compaction interval begins to hurt performance.

Method. Run YCSB+T at seven load fractions of the 4-worker saturation point (18,000 TPS total) using epoch size $E = 1,000$. Each load level is tested with $C \in \{0, 1, 10, 50, 100, 1,000\}$, with no failures injected. The measurement duration is 250 s to ensure $\geq 3,000$ epochs at loads $\geq 70\%$, allowing $C = 1,000$ to trigger multiple times.

Variables.

- **Independent:** Load fraction $\in \{0.50, 0.70, 0.75, 0.80, 0.85, 0.95, 1.00\}$, compaction interval $C \in \{0, 1, 10, 50, 100, 1,000\}$.
- **Dependent:** P95/P99 latency, throughput.
- **Fixed:** Workers = 4, epoch size = 1,000, checkpoint every epoch, 2 client threads, 30 s warmup, 250 s measurement.

5.4.4. Experiment Matrix Summary**Table 5.4:** Compaction experiment matrix (all phases).

Configuration	C	Recovery-only	Failures	Experiments
<i>Phase 1: Steady-state overhead (no failures)</i>				
All 6 configurations	0/1/10/100/1,000/RO	N/A	0	$6 \times 5 = 30$
<i>Phase 2: Recovery performance</i>				
Baseline	0	No	1	5
Compact every checkpoint	1	No	1	5
Compact every 10 checkpoints	10	No	1	5
Compact every 100 checkpoints	100	No	1	5
Compact every 1,000 checkpoints	1,000	No	1	5
Recovery-only	0	Yes	1	5
<i>Phase 3: Load-dependent overhead (no failures)</i>				
$C \in \{0, 1, 10, 50, 100, 1,000\}$ at 7 loads	0/1/10/50/100/1k	No	0	$42 \times 5 = 210$
			Total	270

5.4.5. Analysis Plan

For each research sub-question:

RQ2a: File count vs. restoration time. Scatter plot with linear regression; report the fitted slope and R^2 .

RQ2b: I/O-bound vs. CPU-bound. Compare list and fetch time vs. merge time in the restoration breakdown.

RQ2c: Compaction interval. Median state restoration time with [min–max] ranges by configuration; box plots of steady-state latency; throughput is reported in text.

RQ2d: Continuous vs. recovery-only. Direct comparison of state restoration time and total cost (steady-state overhead + restoration time).

Restoration breakdown. Per-worker state restoration breakdown (list, fetch, merge) by compaction configuration.

5.5. Experiment Suite 3: Rebalance Recovery (RQ3a–RQ3b)

This suite measures rebalancing overhead (RQ3a) and its post-recovery benefit (RQ3b). Loads are fractions of the 4-worker uncoordinated saturation point: 18,000 TPS.

Why $w = 4$. The protocol always pauses workers, redistributes partitions, and restores workers with new assignments (Section 4.5). We vary load relative to saturation rather than worker count. Using 4 workers also permits direct comparison with Suite 2.

5.5.1. Common Configuration

- Checkpointing mode: uncoordinated
- Checkpoint trigger: `epoch_based` (checkpoint every epoch)
- Compaction: enabled at $C = 1$ (compact at every checkpoint), with a minimum cutoff margin $B = 100$ epochs; newer delta files remain uncompactd (Section 4.5.4)
- Workers: 4
- Failed workers: 1
- Epoch size: 1,000 transactions
- Saturation TPS: 18,000 total (9,000 per thread $\times$ 2 threads; from Suite 1, $w = 4$ uncoordinated)

Fault injection. We start the cluster, wait for all workers, and store 200 epochs before killing a worker. This creates enough checkpoint history for recovery.

Controlled variable. Both conditions restart the failed worker. The rebalancing flag is the only difference. Setting `REJOIN_REBALANCE_ENABLED` makes the coordinator rebalance partitions after rejoin; otherwise it keeps the degraded assignment.

5.5.2. Phase 1: Rebalance Overhead (RQ3a)

Method. Run at the 18,000 TPS saturation rate, kill 1 of 4 workers after 200 epochs, and execute the full recovery cycle. We stop input at fault injection, so this phase measures recovery without concurrent work; Phase 2 covers sustained load. Five matched runs omit rebalancing. The coordinator logs the time spent pausing, redistributing, querying storage, sending `InitRecovery`, restoring state, and

finalizing. `time.perf_counter()` records each boundary. Chapter 6 treats wall-clock detection and restart separately.

The *rebalance overhead* is the coordinator-internal total: the stop-the-world window from pausing workers through finalize. ARIA requires all workers to be at the same epoch before reassignment, so workers must pause and the coordinator must determine a recovery point from stable storage. With selective restore (Section 4.5), only workers that gain partitions pay the restore cost, which scales with state size and delta file count.

Metrics.

- Coordinator-internal breakdown: pause, redistribute, storage query, `InitRecovery`, restore, finalize, total

5.5.3. Phase 2: Rebalance Benefit (RQ3b)

Method. Run the workload at multiple fractions of the saturation TPS: {40%, 50%, 70%, 85%}. After 200 epochs, kill 1 of 4 workers. In both conditions the failed worker is restarted; the conditions differ only in the rejoin behavior:

1. **Degraded (no rebalance):** Failed workers restart and rejoin, but the coordinator does *not* trigger rebalance. Surviving workers retain the extra partitions; the rejoined workers sit idle. This isolates the throughput impact of an imbalanced partition distribution.
2. **Rebalance:** Failed workers restart and rejoin, triggering a full cluster-wide recovery that redistributes partitions evenly across all 4 workers. This measures throughput after the rebalance overhead.

The nominal observation window is 90 seconds. It starts after an estimated backlog-drain period: backlog divided by spare capacity, with a safety margin and a 60–300 second limit. Including this wait, analyzed runs last about 121–381 seconds. A second detector checks epoch-rate stability every 10 seconds, but it never triggered within the allowed wait, so every run uses the backlog estimate.

The nominal window covers $[window_start, window_start + 90s]$; the recorded window duration includes the adaptive placement and drain completion described above. Metrics are computed by joining the client-side request log (`client_requests.csv`) with the Kafka output log (`output.csv`) on request ID.

Throughput measurement. Pre-fault throughput is the total input rate. Post-fault throughput is the matched output rate during the observation window. Recovery creates a backlog of roughly input TPS times downtime, so output can briefly exceed input while the backlog drains. Starting the window after the estimated drain makes the post-/pre-fault ratio a measure of sustained capacity.

To complement the windowed measurement, we also report a *steady-state throughput*. Both numbers derive from the same per-second matched-output series within the observation window: post-fault throughput is the median over the full window, while the steady-state number is the median from the detected throughput plateau onward, excluding residual drainage seconds. When both conditions can handle the input, steady-state throughput converges toward the input rate.

Latency measurement. We define two latency cohorts, each measuring a different property:

1. **Steady-state latency** (*completed_in_window*): transactions whose input *and* output both fall within the observation window. This excludes backlog and gives true steady-state latency, but may be empty at high load in the degraded condition, when per-transaction latency exceeds the 90-second window. An empty cohort indicates that the system cannot sustain the offered load.
2. **End-to-end latency** (*outputs_in_window*): all transactions completing during the window, regardless of when they were submitted. This includes queue delay from the backlog and is always available when outputs exist.

The benefit comparison (Table 6.15) reports the end-to-end cohort exclusively: it is always available, includes the queuing delay in which the two conditions differ, and is the authoritative P95 path in the analysis builder. Completed-in-window P95 remains available as a secondary diagnostic.

Metrics.

- Pre-fault throughput
- Post-fault throughput
- Steady-state throughput
- Full-window throughput ratio
- Steady-state ratio: steady-state / pre-fault
- P95 end-to-end latency
- Consistency verification (exactly-once output, money conservation), sourced from the persisted raw Phase 2 records

Break-even analysis. If rebalancing provides a sustained throughput advantage, we define the *break-even time* as the point where the cumulative throughput of the rebalanced condition overtakes the degraded condition (which starts ahead because rebalancing incurs recovery overhead). Formally, $\Delta(t) = \sum_{i=0}^t (TPS_{\text{rebalance}}(i) - TPS_{\text{degraded}}(i))$; break-even is the first t where $\Delta(t) \geq 0$ after having been negative. Break-even is only reported when the rebalanced steady-state ratio exceeds the degraded ratio by at least 2%. If both conditions sustain the input rate, no break-even exists. The reported search is limited to the nominal 90 s post-recovery observation window.

Load levels. With 4 workers and 1 failure, one survivor inherits a second partition and doubles its pre-fault load. The other two survivors keep one partition each. The expected tipping point is therefore 50% total load, where the overloaded survivor reaches its estimated capacity. Table 5.5 lists the tested rates.

Table 5.5: Load levels and survivor utilization for 1-failure ($f = 1, w = 4$). Input TPS is total across all client threads.

Load level	Input TPS	Survivor load	Hypothesis
40%	7,200	80%	No throughput benefit
50%	9,000	100%	Tipping point
70%	12,600	140%	Degraded overloaded
85%	15,300	170%	Degraded severely overloaded

5.5.4. Experiment Matrix Summary

Table 5.6 summarizes the 50 runs in this suite. Phase 1 isolates the cost of rebalancing by comparing matched runs with and without it at saturation. Phase 2 then tests whether rebalancing recovers that cost at four load levels. Each configuration has five repetitions, supporting the median and range comparisons reported in Section 6.3.

Table 5.6: Rebalance experiment matrix ($w = 4$, saturation TPS = 18,000).

Phase	Variable	Values	Experiments
P1: Overhead	Condition	{Deg., Reb.} ($f = 1$)	$2 \times 5 = 10$
P2: Benefit	Load $\times$ Condition	{40, 50, 70, 85}% $\times$ {Deg., Reb.}	$4 \times 2 \times 5 = 40$
Total			50

5.6. Scalability Analysis Framework

5.6.1. Amdahl's Law Fitting

To summarize the observed end-to-end scaling, we fit the committed-throughput data to Amdahl's Law [3]:

$$\text{Speedup}(n) = \frac{1}{S + \frac{1-S}{n}} \quad (5.2)$$

Here n is the scaling factor and S the effective non-scaling fraction. We fit S to Phase 1 committed throughput with `scipy.optimize.curve_fit`. Because STYX requires at least 2 workers, $w = 2$ is the baseline. The fitted S is therefore relative to two workers and is not directly comparable with fits that start at one. It is a descriptive aggregate of the measured setup rather than a direct measurement of barrier cost: barriers, epoch-boundary work, single-node resource contention, and client pacing may all contribute. Under the model's fixed- S assumption, $1/S$ is an optimistic asymptote; overhead that grows with worker count can produce an earlier plateau.

5.6.2. Statistical Reporting

The original STYX, ARIA, and comparable SFaaS evaluations report point estimates from a small number of runs without significance tests [28, 25, 36, 19]. We follow that convention: up to 5 repetitions, medians, and [min-max] ranges where spread matters. This convention fits two properties of systems benchmarks: run-to-run variability is small (the coefficient of variation of committed throughput is below 5% in our experiments), and the effects of interest are large (order-of-magnitude differences in latency and restoration time), so repetitions and ranges convey reliability directly. We also prefer medians and quantiles over means throughout, as recommended for skewed performance data [17].

We do not use significance tests as primary evidence: with $n = 5$, they have little power for small effects, while the main effects have non-overlapping ranges. When a difference is smaller than run-to-run variation, we say so instead of claiming equivalence.

5.7. Threats to Validity

Internal validity. The binary search saturation algorithm assumes a monotonic relationship between input TPS and P95 latency, which holds at steady state but may not hold during transients; the warmup period mitigates this. Experiments within each suite run in a fixed order (e.g., coordinated

before uncoordinated, ascending worker count); Docker image caching and OS page-cache effects could systematically advantage later configurations. Randomized ordering would provide stronger guarantees, though 5 repetitions and warmup partially mitigate order effects.

External validity. We evaluate using only the YCSB+T workload with 10,000 keys. Results may differ for workloads with different transaction complexity and key distributions (e.g., TPC-C [24], DeathStarBench [14]). We chose YCSB+T because it is supported by all baseline systems and provides controlled contention.

Deployment validity. All containers run on one machine with loopback networking. This removes network variation but understates distributed communication cost. Local MinIO GETs take about 1.7 ms, far below the 100–200 ms documented for small Amazon S3 objects¹. Production recovery would therefore take longer. Relative comparisons remain useful because every configuration uses the same environment.

Resource contention. With up to 12 workers, 3 Kafka brokers, 4 MinIO instances, a coordinator, and client threads all sharing a single machine, CPU and memory contention between containers could introduce artifacts not present in a distributed deployment. Single-threaded workers limit each worker container’s CPU demand to roughly one core, which mitigates CPU interference, but shared memory bandwidth and I/O channels remain a potential confound.

Client generation capacity. The workload generator is two Python client processes that pace sends with fixed sleep slices. The realized send rate is not validated against the nominal rate in the recorded aggregates, so at the highest worker counts part of the offered-vs-committed gap in Figure 6.1 could reflect client-side pacing shortfall rather than barrier queuing alone. The mode comparison is unaffected: both modes are driven by the same client at the same nominal rates, and the binary search observed failing probes above each accepted saturation point, so server-side queuing is present at the margin.

Construct validity. The $P95 < 1,000$ ms saturation threshold is a pragmatic choice. Different thresholds would shift absolute saturation points, but the relative comparison between modes would remain valid. The recorded search probes let us check this directly: re-deriving each configuration’s saturation point at thresholds of 750, 1,500, and 2,000 ms shifts the absolute anchors but never separates the modes by more than four search steps, with neither mode winning consistently.

Reproducibility. All experiment configurations, analysis scripts, and deployment scripts are included in the repository under `experiments/runners/` and `scripts/`.

¹<https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html>

6

Results

This chapter presents experimental results for the three suites described in Chapter 5: scalability (RQ1a–RQ1d), compaction and recovery (RQ2a–RQ2d), and rebalance recovery (RQ3a–RQ3b).

6.1. Scalability Results (RQ1a–RQ1d)

6.1.1. Phase 1: Saturation Points

Table 6.1 shows that coordinated and uncoordinated checkpointing reach similar saturation points at every worker count. The difference changes direction across configurations and never exceeds two 1,000 TPS search steps, so neither mode has a consistent performance advantage. Saturation is the highest tested input rate whose median P95 stays below 1,000 ms across 5 repetitions.

Table 6.1: Saturation throughput by worker count and checkpointing mode (median of 5 repetitions). Saturation TPS is the total input rate found by binary search with 1,000 TPS steps; Throughput is the measured committed TPS at saturation. The coefficient of variation of committed throughput is below 5% for every configuration. The modes differ by at most two search steps at each worker count, and neither is consistently higher.

Workers	Mode	Sat. TPS	Throughput	[Min–Max]	P95 (ms)	P99 (ms)
2	Coord.	11,000	10,745	[10,521–10,745]	296	369
2	Uncoord.	11,000	10,978	[10,745–10,978]	526	643
4	Coord.	19,000	17,653	[17,300–17,653]	886	1,002
4	Uncoord.	18,000	16,735	[16,079–16,735]	957	1,186
6	Coord.	25,000	21,019	[20,637–21,415]	614	674
6	Uncoord.	26,000	21,852	[21,852–22,264]	645	727
8	Coord.	32,000	25,000	[24,577–25,439]	787	982
8	Uncoord.	34,000	26,552	[26,102–27,018]	900	1,061
10	Coord.	38,000	28,197	[27,742–28,667]	757	828
10	Uncoord.	38,000	28,667	[27,302–28,667]	771	850
12	Coord.	44,000	31,094	[30,616–31,588]	844	927
12	Uncoord.	46,000	32,000	[31,516–32,500]	914	1,010

Measured throughput falls from 98–100% of input at $w = 2$ to 70–71% at $w = 12$. More workers increase time spent waiting at the five per-epoch ARIA barriers because each barrier moves at the pace of the slowest worker. Client pacing may also contribute at the largest scales (Section 5.7). Both modes scale from 11,000 TPS at $w = 2$ to 44,000–46,000 TPS at $w = 12$ (Figure 6.1).

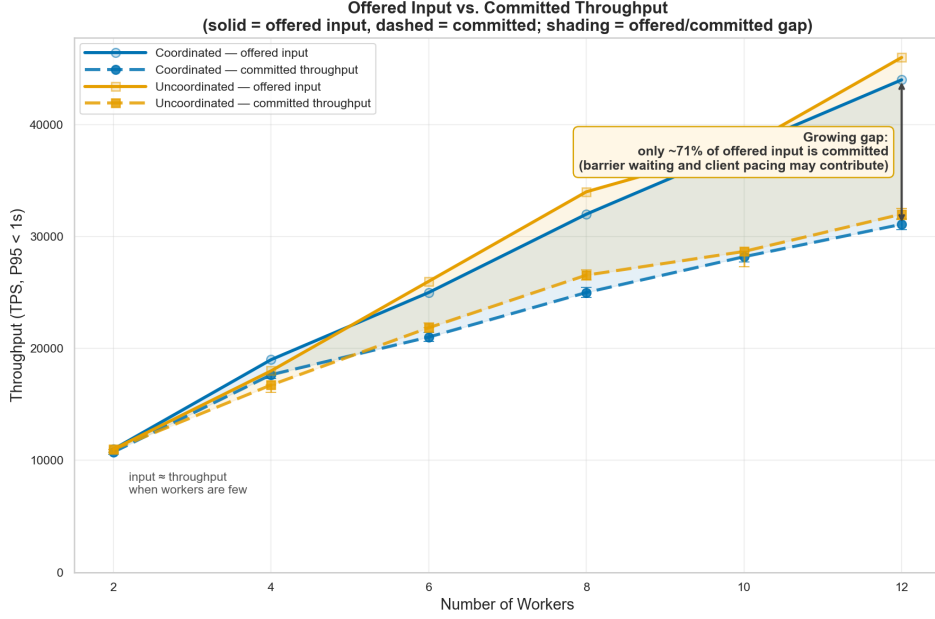


Figure 6.1: Offered input rate (solid) vs. committed throughput (dashed) by worker count. The shaded band is the gap between the two: work that is accepted but cannot be committed within the latency budget. Barrier waiting contributes to this gap, although client pacing may also contribute at the largest scales. The gap widens with scale (at $w = 12$ only ~71% of the offered input is committed), illustrating that committed throughput cannot follow the offered rate indefinitely as workers are added. Error bars on committed throughput show the min–max across the 5 repetitions (the spread is small: coefficient of variation <5%).

Mode gap (RQ1c). Uncoordinated saturation TPS differs from coordinated by -5.3% to $+6.3\%$, with a median of $+2.0\%$. The sign changes across worker counts, and every gap is at most two 1,000 TPS search steps (Figure 6.2). In Phase 1, coordinated mode uses its 10-second time-based trigger, while uncoordinated mode uses the epoch intervals defined in Section 5.3. The separate strategy sweep shows how changing that cadence affects performance (Table 6.5).

6.1.2. Scalability Analysis (RQ1a)

Table 6.2 shows that both modes lose about half their per-worker efficiency by $w = 12$. Their efficiency remains nearly identical, so the scaling loss comes from shared execution costs rather than checkpointing mode. The calculation uses committed throughput (transactions fully processed within the measurement window) and $w = 2$ as the baseline.

Table 6.2: Scaling efficiency relative to the $w = 2$ baseline (committed throughput). Both modes lose roughly half their per-worker efficiency by $w = 12$, and the two track each other closely. The shared loss therefore cannot be attributed to the checkpointing mode.

Workers	Coordinated			Uncoordinated		
	TPS	Speedup	Eff.	TPS	Speedup	Eff.
2	10,745	1.00×	100%	10,978	1.00×	100%
4	17,653	1.64×	82%	16,735	1.52×	76%
6	21,019	1.96×	65%	21,852	1.99×	66%
8	25,000	2.33×	58%	26,552	2.42×	60%
10	28,197	2.62×	52%	28,667	2.61×	52%
12	31,094	2.89×	48%	32,000	2.91×	49%

Amdahl’s Law describes scaling under a fixed serial fraction S . We fit $\text{speedup}(n) = 1/(S + (1 - S)/n)$

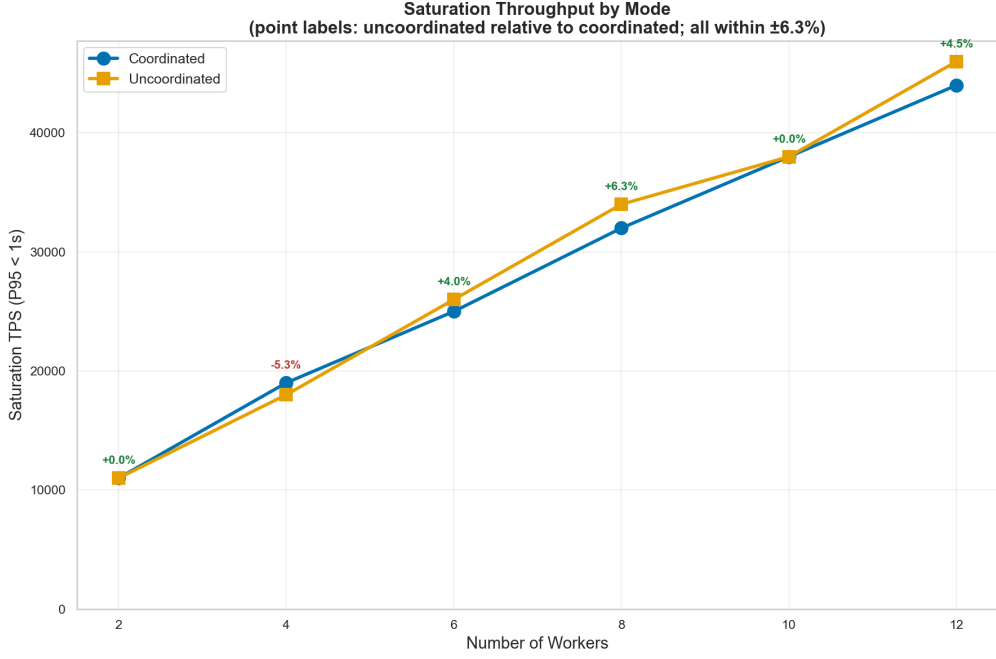


Figure 6.2: Saturation throughput by mode across worker counts. The two modes track each other closely; inline labels give the percentage difference (uncoordinated vs. coordinated), which fluctuates within $\pm 6.3\%$ with no clear systematic trend.

to committed throughput, where n is the scaling factor relative to $w = 2$. The fit yields:

- **Coordinated:** effective non-scaling fraction $S = 22.6\%$, $R^2 = 0.99$, fitted asymptote = $4.4\times$.
- **Uncoordinated:** effective non-scaling fraction $S = 22.3\%$, $R^2 = 0.99$, fitted asymptote = $4.5\times$.

Figure 6.3 compares the observed speedups with these fitted curves. The fitted values are conditional model outputs, not demonstrated capacity limits. They are relative to the two-worker baseline: at $w = 12$, the observed speedup is already $2.89\text{--}2.91\times$, so the fitted model permits only about $1.5\times$ further improvement even with unbounded workers. The fitted fraction aggregates all non-scaling effects in the measured setup, including barriers, epoch-boundary work, single-node resource contention, and client pacing. If coordination overhead grows with worker count, throughput can plateau before the fitted asymptote. Because client pacing may also contribute, the fit does not isolate a Sryx-only limit.

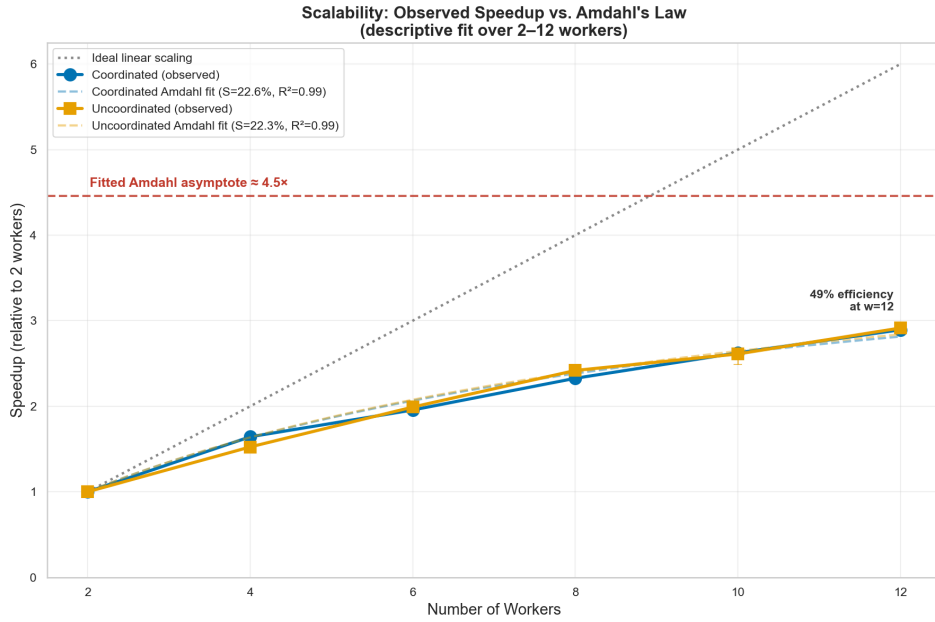


Figure 6.3: Observed speedup vs. Amdahl’s Law fit (committed throughput). Both modes exhibit sub-linear scaling with nearly identical effective non-scaling fractions ($S \approx 22\%$), reaching only 48–49% efficiency at $w = 12$. The dotted line is ideal linear scaling; the red line marks the fitted asymptote ($1/S$; $4.4\times$ coordinated, $4.5\times$ uncoordinated) under the fixed- S assumption. It is not an observed capacity limit. Error bars show the min–max committed throughput across the 5 repetitions, propagated to the speedup.

Per-worker committed throughput decreases from $\sim 5,400$ TPS at $w = 2$ to $\sim 2,600$ – $2,700$ TPS at $w = 12$, a $\sim 50\%$ drop.

6.1.3. Phase 2: Epoch Size Impact (RQ1b)

Phase 2 tests seven epoch sizes at each Phase 1 saturation rate. Epoch size 1 times out at every worker count because each transaction pays the cost of all five ARIA barriers. We therefore report sizes $\{5, 50, 100, 250, 500, 1,000, 2,000\}$, including the Phase 1 default.

Table 6.3: Epoch size impact on P95 latency (median across 6 worker counts, at saturation TPS). Status is defined by the P95 threshold: “Met” = $P95 < 1,000$ ms, “Over” = $P95 \geq 1,000$ ms. In this aggregate, epoch size 100 has 13–14 $\times$ lower tail latency than the default of 1,000; very small (≤ 5) or very large ($\geq 2,000$) epochs perform far worse. **Bold** marks the lowest aggregate median.

Epoch Size	Coord. P95 (ms)	Uncoord. P95 (ms)	Status
1	N/A	N/A	Failed (timeout)
5	169,187	169,164	Over ($>100\times$)
50	570	680	Met
100	59	58	Met (best tested)
250	94	100	Met
500	188	215	Met
1,000	772	836	Met (baseline)
2,000	7,385	6,519	Over

Epoch size has a non-linear effect on tail latency. Sizes ≤ 5 exceed 150 seconds because each small batch still pays the fixed cost of conflict detection and five barriers. At size 2,000, larger conflict sets and more fallback re-execution raise P95 above 6 seconds. These overloaded runs mainly measure a growing queue, so only their ordering is reliable. Across the six worker counts, size 100 gives the lowest median P95 (58–59 ms), 13–14 $\times$ below the default size of 1,000. The per-worker results are not uniform:

at $w = 2$, size 250 is lowest in both modes (67–71 ms), while size 100 is lowest at $w = 4$ through $w = 12$. Sizes 100–250 therefore form the best tested range. Later suites keep the 1,000 default for comparison with the original system.

As shown in Figure 6.4, the same broad pattern appears at all six worker counts: P95 falls sharply between sizes 5 and 100–250, then rises sharply at 2,000. The difference between the extremes spans two orders of magnitude and is much larger than the run-to-run spread within a configuration. No single worker count drives the aggregate result.

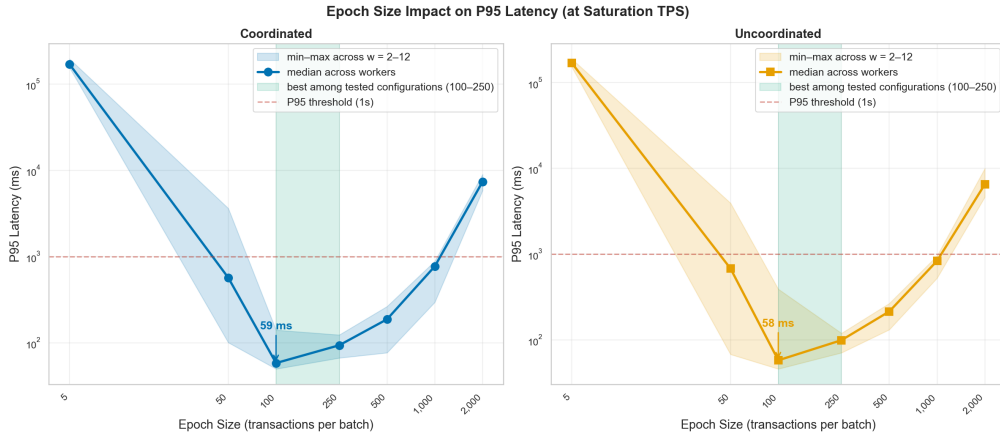


Figure 6.4: P95 latency vs. epoch size at saturation TPS. The line is the median across worker counts and the shaded band is the min–max across $w = 2$ –12. Both modes show a sharp drop from epoch size 5 to the best tested range at 100–250 and a sharp rise at 2,000. Size 100 has the lowest aggregate median; size 250 is lowest at $w = 2$.

6.1.4. Phase 3: Checkpoint Strategy Sweep (RQ1c)

Phase 3 tests nine checkpoint strategies (plus the Phase 1 baseline for comparison) across all six worker counts for uncoordinated mode only (coordinated mode supports only its built-in time-based trigger, whose interval is configurable but whose strategy is not; Section 2.4). We first present detailed results for $w = 2$ (Table 6.4), then show how the pattern scales (Table 6.5).

Table 6.4 shows the checkpoint strategy sweep for $w = 2$ at the Phase 1 saturation TPS of 11,000. Throughput is stable across all strategies (96–100% of the 11,000 TPS input rate), confirming that checkpoint frequency does not limit throughput at saturation. However, tail latency spans an order of magnitude from the most aggressive strategy (epoch_1, 2,848 ms) to the plateau (epoch_100+, ~270 ms).

Each checkpoint adds serialization and a storage PUT. At saturation, the system has little spare capacity for this work. P95 falls sharply as the interval grows to 100 epochs, then levels at 263–276 ms for $w = 2$. The Phase 1 interval of 55 epochs yields 526 ms; intervals of at least 100 roughly halve it.

Scaling across worker counts. Table 6.5 extends the analysis to four representative worker counts. The pattern observed at $w = 2$ holds across scales: frequent checkpointing (epoch_1) produces extreme tail latency (2,848–15,178 ms P95), while strategies with intervals ≥ 100 epochs converge to a plateau below 1,000 ms at all worker counts tested. The critical threshold shifts with scale: at $w = 2$, even epoch_100 meets the $P95 < 1,000$ ms criterion, while at $w = 12$, strategies with intervals < 50 epochs exceed it.

Throughput remains insensitive to checkpoint frequency across all worker counts: median committed TPS varies by less than 8% across strategies (excluding epoch_1, which reduces throughput by up to 19%). Among the plateau strategies (epoch_100, epoch_250, epoch_500), the differences in median P95 (263–276 ms at $w = 2$) are smaller than the run-to-run spread within any single strategy, so we treat them

Table 6.4: Checkpoint strategy sweep ($w = 2$, uncoordinated, 11,000 TPS target). `epoch_N` = checkpoint every N epochs; `time_10s` = epoch-aligned time-based trigger (~ 10 s). The Phase 1 baseline (`epoch_55`, targeting ~ 10 s) is included for comparison. P95 plateaus once the interval reaches 100 epochs; checkpointing more often inflates tail latency and buys no throughput. **Bold** marks the plateau rather than a single minimum whose difference is within run-to-run variation.

Strategy	P95 Latency (ms)	Throughput (TPS)
<code>epoch_1</code>	2,848	10,521
<code>epoch_5</code>	1,493	10,978
<code>epoch_10</code>	882	10,978
<code>epoch_25</code>	373	10,745
<code>epoch_50</code>	361	10,745
P1 default [†]	526	10,978
<code>epoch_100</code>	276	10,745
<code>epoch_250</code>	263	10,745
<code>epoch_500</code>	273	10,745
<code>time_10s</code>	383	10,745

[†]Phase 1 baseline: `epoch_based` trigger with an interval computed to target ~ 10 s between checkpoints (55 epochs for $w = 2$; varies by worker count).

Table 6.5: Checkpoint strategy sweep across worker counts: median P95 latency in ms (uncoordinated, at Phase 1 saturation TPS, median of 5 repetitions). Saturation TPS is shown in parentheses. Intervals of at least 100 epochs form a stable low-latency range at every worker count. **Bold** marks this range.

Strategy	$w = 2$ (11,000)	$w = 4$ (18,000)	$w = 8$ (34,000)	$w = 12$ (46,000)
<code>epoch_1</code>	2,848	7,655	13,042	15,178
<code>epoch_5</code>	1,493	1,236	4,728	2,420
<code>epoch_10</code>	882	712	2,277	1,227
<code>epoch_25</code>	373	516	1,227	1,062
<code>epoch_50</code>	361	476	903	953
<code>epoch_100</code>	276	426	902	897
<code>epoch_250</code>	263	372	781	844
<code>epoch_500</code>	273	380	899	786
<code>time_10s</code>	383	507	857	901

as indistinguishable. The `time_10s` strategy, which implements Algorithm 1, performs comparably to `epoch_50`–`epoch_100`. Alignment itself therefore costs little, making this strategy a practical option when predictable wall-clock spacing matters.

Cadence comparison. At $w = 2$ ’s saturation epoch rate (~ 5 epochs per second, since each worker batches up to 1,000 of its own transactions per epoch), the plateau strategies (`epoch_100`+) correspond to wall-clock intervals of roughly 18 s and longer. Phase 1’s default (`epoch_55`) fired every ~ 10 s, matching the intended target (Chapter 5). It lies outside the steep-latency region of `epoch_1`–`epoch_10`, although its P95 is roughly double the plateau’s.

Recovery trade-off and compaction. Longer intervals reduce normal-operation cost but increase replay after failure. The compaction suite appears to disagree: checkpointing and compacting every epoch adds only 37% P95 there, while `epoch_1` causes severe latency degradation here. Load explains the difference. The compaction suite has spare capacity; the saturation sweep does not. At saturation, intervals of at least about 50 epochs are advisable, with 100 or more in the lowest-latency tested range.

6.1.5. Phase 4: Contention Impact (RQ1d)

Phase 4 evaluates the impact of key-access skew on throughput, latency, and consistency. Each configuration is offered its Phase 1 saturation TPS (the maximum sustainable throughput under uniform access) while the key distribution follows a Zipfian distribution with skew parameter z . The uniform baseline ($z = 0.0$) is reused from Phase 1; we test $z \in \{0.25, 0.5, 0.75, 0.99\}$ across both modes.¹

Throughput degradation. We define the *throughput ratio* as committed TPS at skew level z divided by committed TPS at $z = 0.0$ for the same (worker, mode) configuration. Table 6.6 shows this ratio for coordinated mode (uncoordinated ratios are within 6 percentage points and omitted for brevity). Contention has no measurable impact at $z = 0.25$ and $z = 0.5$ (coordinated ratios ≥ 0.96). At $z = 0.75$, throughput degrades sharply and the degradation *worsens* with worker count: from 13% loss at $w = 2$ to 55% at $w = 12$. At $z = 0.99$, throughput collapses to 13–42% of baseline across tested configurations ($w = 2$ –8, coordinated), with absolute committed TPS of only 3,333–4,509.

Table 6.6: Committed TPS under Zipfian skew (coordinated; median of 5 repetitions). Parentheses give the ratio to $z = 0$: $\approx$ marks a ratio ≥ 0.95 , $\downarrow$ marks degradation, and TO denotes a timeout.

Workers	$z = 0.0$	$z = 0.25$	$z = 0.5$	$z = 0.75$	$z = 0.99$
2	10,745	10,745 $\approx$ (1.00)	10,745 $\approx$ (1.00)	9,352 $\downarrow$ (0.87)	4,509 $\downarrow$ (0.42)
4	17,653	17,653 $\approx$ (1.00)	16,961 $\approx$ (0.96)	12,014 $\downarrow$ (0.68)	4,199 $\downarrow$ (0.24)
6	21,019	21,019 $\approx$ (1.00)	21,827 $\approx$ (1.04)	14,012 $\downarrow$ (0.67)	3,927 $\downarrow$ (0.19)
8	25,000	24,577 $\approx$ (0.98)	24,577 $\approx$ (0.98)	15,263 $\downarrow$ (0.61)	3,333 $\downarrow$ (0.13)
10	28,197	28,197 $\approx$ (1.00)	28,197 $\approx$ (1.00)	14,701 $\downarrow$ (0.52)	TO
12	31,094	31,094 $\approx$ (1.00)	31,588 $\approx$ (1.02)	14,114 $\downarrow$ (0.45)	TO

At $z = 0.75$, throughput levels off at about 14,000–15,300 TPS beyond $w = 6$. Adding workers no longer helps because hot keys create cross-partition conflicts and more ARIA re-execution. The longer fallback phase cancels the scaling gains seen under uniform access.

Latency impact. Latency degrades in tandem with throughput (Table 6.7). At $z = 0.25$, P95 latency remains comparable to the uniform baseline (0.28–0.83 s vs. 0.30–0.89 s). At $z = 0.5$, P95 exceeds 1 s at most worker counts (1.4–2.2 s), as the additional ARIA re-execution rounds imposed by cross-partition conflicts begin to dominate. At $z = 0.75$, P95 grows from 9.4 s ($w = 2$) to 79.7 s ($w = 12$), confirming that the conflict-driven overhead compounds with scale. At $z = 0.99$, P95 reaches 66–447 s, rendering the system unresponsive. Beyond $z = 0.5$ the system is overloaded, so these P95 values largely measure a growing queueing backlog over the measurement window (cf. Section 6.3.2). The ordering across skew levels is therefore more informative than the absolute values.

Validation under extreme contention. All experiments at $z \leq 0.75$ reached quiescence and passed consistency verification with zero wrong values across both modes and all worker counts. At $z = 0.99$, the final-state checker reported 40–342 differing key values in some $w = 4$ –8 runs, while $w = 10$ and $w = 12$ timed out before collecting sufficient output. These mismatches are not sufficient to establish serialization violations. The checker waits at most 120 seconds after a 30-second run, yet measured P95 latency at $z = 0.99$ ranges from 66 to 447 seconds. It also constructs the expected balances from every submitted transfer, including transactions that may still be queued when validation begins (Section 5.3). The comparison can therefore place a partially processed state against a fully submitted workload.

¹ $z = 0.99$ at $w = 10$ and $w = 12$ timed out before sufficient outputs could be collected (29% completion at $w = 10$). All other configurations completed 5 repetitions.

Table 6.7: P95 latency in seconds under Zipfian skew (coordinated; median of 5 repetitions). ↑ marks $P95 \geq 1$ s; “—” denotes a timeout.

Workers	$z = 0.0$	$z = 0.25$	$z = 0.5$	$z = 0.75$	$z = 0.99$
2	0.30	0.28	0.96	↑ 9.4	↑ 66
4	0.89	0.83	↑ 2.2	↑ 20.9	↑ 155
6	0.61	0.64	↑ 1.4	↑ 28.3	↑ 273
8	0.79	0.69	↑ 1.8	↑ 42.0	↑ 447
10	0.76	0.73	↑ 1.6	↑ 58.1	—
12	0.84	0.81	↑ 1.5	↑ 79.7	—

At $z = 0.99$, a few hot keys serialize much of the work. Service rate falls to 13–42% of the uniform baseline while input stays fixed, so the queue cannot drain. The result marks a capacity boundary for both modes; it does not isolate a checkpointing or ARIA defect. A correctness test at this skew needs a lower, skew-specific input rate and an uncapped drain to quiescence (Section 7.4.5).

Mode comparison under contention. The gap between the two modes remains small (within 6 percentage points of throughput ratio) across all contention levels, extending the Phase 1 finding (Section 6.1) to skewed workloads. Both modes execute the same ARIA conflict detection, so contention affects them equally. At higher skew, slower epoch progress also reduces the frequency of the fixed epoch-based trigger. The mode comparison in this range therefore provides stronger evidence about shared ARIA execution than about checkpoint-persistence overhead.

6.2. Compaction and Recovery Results (RQ2a–RQ2d)

We test six compaction configurations with five repetitions and one worker failure: 30 recovery experiments plus a 210-run load sweep. Unless stated otherwise, results are medians. The interval C counts stored checkpoints. Although the trigger runs every epoch, the busy guard skips roughly every other attempt. The baseline stores about 1,460 files per partition across 3,200 epochs, so C checkpoints span about $2C$ epochs in this suite.

6.2.1. State Restoration Time by Configuration

Table 6.8 presents state restoration time and file counts across all six configurations. State restoration time measures only the stable storage I/O phase of the first recovery: listing, fetching, and merging checkpoint files. It excludes fault detection, worker restart, Kafka replay, and the subsequent rebalance recovery (Table 6.13). When multiple workers recover simultaneously, we report the maximum (bottleneck). Each worker fetches only its own partitions’ `data/*.bin` files.

Restoration ranges from 43 ms at $C = 1$ to 4,971 ms without compaction.² The no-compaction baseline is not the original Sryx policy; original Sryx compacts after every snapshot, closest to $C = 1$. We include $C = 0$ only to isolate frequency. The load sweep shows that the original always-on policy is a poor default above about 50% load (Section 6.2.5).

The ranges do not overlap. Every $C = 1$ and $C = 10$ run restores in 40–75 ms, versus 4,785–5,265 ms for the baseline. Even the slowest $C = 100$ run is about $9\times$ faster than the fastest baseline run. $C = 1$ is

²These runs predate the $B = 100$ cutoff margin added for rebalancing (Section 4.5.4). They use only the $C + 1$ margin. With $B = 100$, configurations below $C = 100$ would leave delta files from up to roughly 100 recent epochs uncompacted and could fetch up to 100 more files per partition.

Table 6.8: State restoration time and file counts by compaction configuration (median of 5 repetitions; 1 failure). Restoration time falls with the number of files fetched; $C = 1$ fetches the fewest files and restores fastest (**bold**).

Configuration	Restore (ms)	[Min–Max] (ms)	Files Fetched	Files in MinIO
Baseline	4,971	[4,785–5,265]	2,988	5,841
$C = 1$	43	[40–49]	7	19
$C = 10$	63	[54–75]	27	54
$C = 100$	474	[357–509]	257	489
$C = 1,000$	2,941	[2,695–3,485]	1,726	3,386
Recovery-only	4,998	[4,612–5,118]	3,040	5,954

Restore (ms): time to list, fetch, and merge checkpoint files from stable storage (max across recovering workers). *Files Fetched:* data/* .bin files loaded by the bottleneck worker during state restoration (each worker only fetches files for its assigned partitions). *Files in MinIO:* total data/* .bin files across all partitions at time of fault injection.

consistently faster than $C = 10$ (43 vs. 63 ms median).

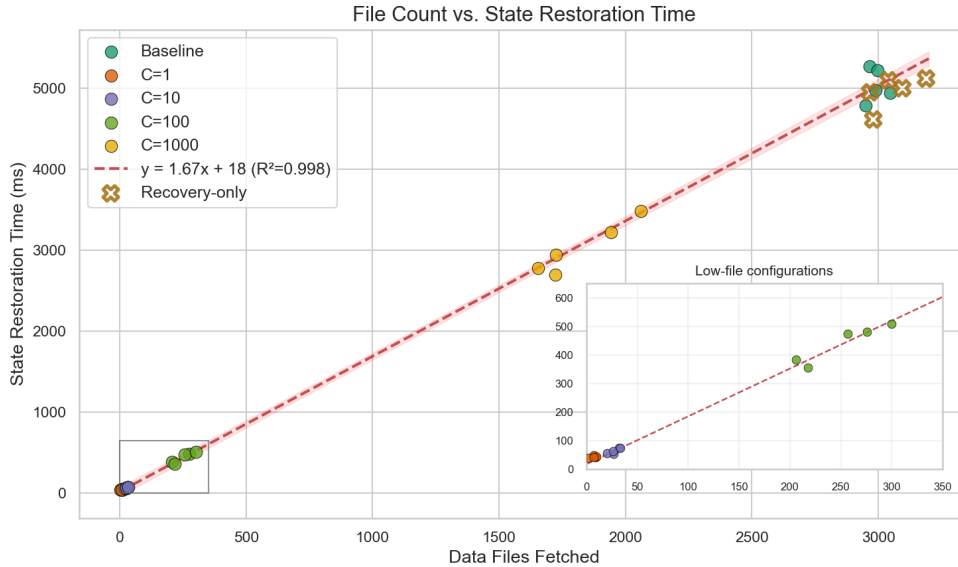
6.2.2. File Count–Restoration Time Correlation (RQ2a)

Linear regression on the 25 non-recovery-only experiments (5 configurations $\times$ 5 repetitions) yields:

$$T_{\text{restore}} = 1.67 \cdot n_{\text{files}} + 18 \quad (R^2 = 0.998) \quad (6.1)$$

The near-perfect fit indicates a linear relationship (Figure 6.5). The slope of 1.67 ms per file represents the combined per-file stable-storage overhead in this recovery path, including the growth in listing and fetching work. The 18 ms intercept represents the fitted fixed component; the experiment does not separate connection setup from other fixed work. Individual runs range from 2 files ($C = 1$) to 3,047 files (baseline), spanning 40 ms to 5,265 ms. Medians are 7 files ($C = 1$) and 2,988 files (baseline).³

This linearity matches the cost model: because merge time is negligible (Section 6.2.3), state restoration time is approximately a linear function of file-related stable-storage I/O.

**Figure 6.5:** Restoration time versus files fetched. The fit and descriptive 95% confidence band use the 25 continuous-compaction runs; recovery-only runs are excluded from the fit. The inset enlarges configurations below 350 files.

³The 25 observations cluster by compaction configuration. The high R^2 partly reflects wide separation between groups, but a regression on the 5 group medians yields the same slope (1.67 ms/file), confirming the relationship holds.

6.2.3. I/O vs. CPU Breakdown (RQ2b)

Table 6.9 and figure 6.6 decompose state restoration time into three phases: object listing, data fetching, and in-memory merging. Recovery is almost entirely I/O-bound, with the I/O ratio exceeding 99% in every configuration. The merge operation (deserialization and `dict.update()`) takes less than 10 ms even for the baseline’s 2,988 files. Within the I/O component, fetching dominates listing by roughly 5:1 in the baseline (4,208 ms vs. 788 ms).

Table 6.9: Median state-restoration time by phase in milliseconds. Stable-storage I/O accounts for more than 99% of restoration time in every configuration (**bold**); in-memory merging is negligible.

Configuration	List (MinIO)	Fetch (MinIO)	Merge (CPU)	I/O Ratio
Baseline	788	4,208	6.5	99.9%
C = 1	23	17	0.3	99.3%
C = 10	24	47	0.4	99.5%
C = 100	67	416	1.1	99.8%
C = 1,000	536	2,393	5.3	99.8%
Recovery-only	759	4,293	9.0	99.8%

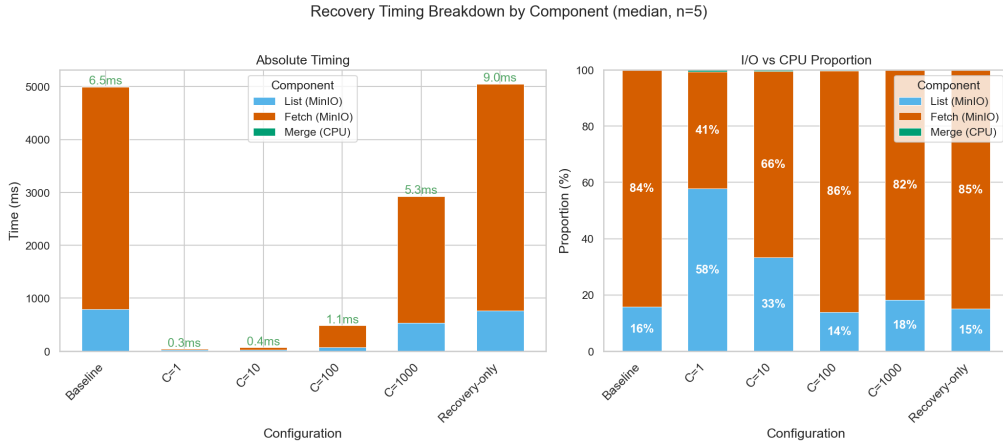


Figure 6.6: State-restoration timing by phase: absolute duration (left) and normalized proportion (right).

6.2.4. Compaction Interval Trade-off (RQ2c)

At 11% load, Table 6.10 compares the restoration benefit of compaction with its steady-state cost. Among the continuous settings, $C = 1$ has both the fastest restoration and the lowest P95. The later load sweep tests whether this result holds at higher utilization.

Table 6.10: Restoration and steady-state performance by compaction configuration at 11% load. **Bold** marks the best compaction-enabled value; ↓ marks improvement and ↑ marks degradation relative to baseline.

Config.	Restore (ms)	vs. BL	TPS	Avg Lat. (ms)	P95 (ms)	P99 (ms)
Baseline	4,971	N/A	2,022	26.6	43	69
Recovery-only	4,998	+0.5%	2,022	24.7 ≈ (−7%)	40	65
C = 1	43	↓ −99.1%	2,026	37.2 ↑ +40%	59	76
C = 10	63	↓ −98.7%	2,026	41.2 ↑ +55%	70	85
C = 100	474	↓ −90.5%	2,031	46.4 ↑ +74%	130	179
C = 1,000	2,941	↓ −40.8%	2,031	53.8 ↑ +102%	244	404

Throughput is stable across all configurations (<0.5% spread). Latency shows a different pattern:

less frequent compaction (larger C) produces higher steady-state latency, because each compaction event processes more accumulated files in a single batch. The baseline and recovery-only configurations share the same code path, achieving the lowest latency (P95 = 43 and 40 ms). Among compaction-enabled configurations, $C = 1$ has the lowest overhead (P95 = 59 ms), while $C = 1,000$ produces the worst tail latency (P99 = 404 ms, $\sim 6\times$ baseline) because large batch sizes create I/O bursts that compete with transaction processing.

$C = 1$ offers the best trade-off among compaction-enabled configurations: the fastest state restoration (43 ms, a 99.1% reduction) at a +37% P95 premium. It dominates $C = 10$ on *both* axes (faster restoration *and* lower latency), making $C = 10$ strictly inferior at this load level (this latency ranking reverses at high utilization; see Section 6.2.5).

Figure 6.7 presents the latency distributions (throughput, being flat across configurations, is reduced to a note on the figure). Figure 6.8 plots restoration time against steady-state latency: the choice at this load is between two corners, and every intermediate interval is dominated.

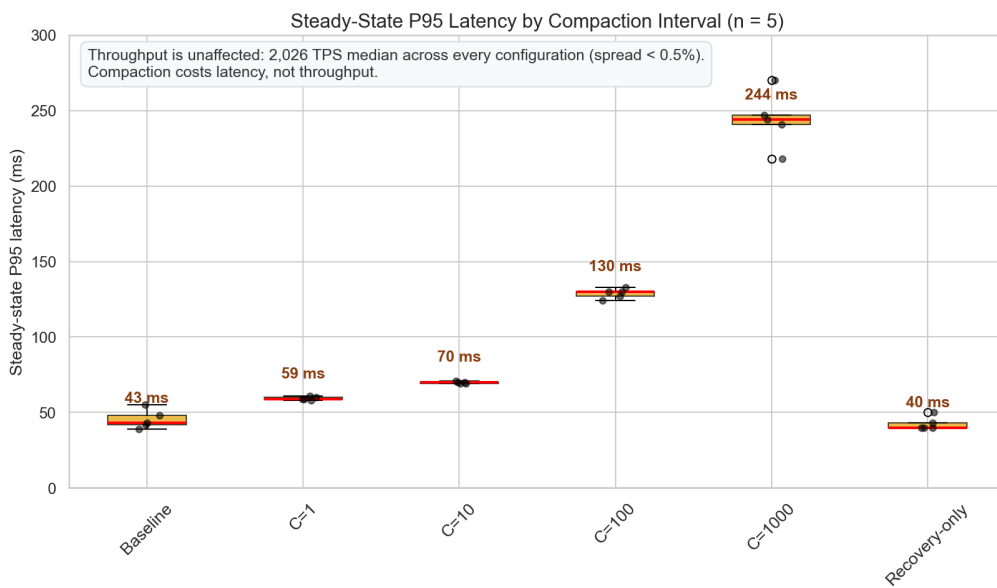


Figure 6.7: Steady-state P95 latency by compaction configuration over 5 repetitions; labels show medians. Throughput spans 2,022–2,031 TPS.

6.2.5. Load-Dependent Compaction Overhead (RQ2c cont.)

The preceding analysis measured compaction overhead at $\sim 11\%$ of saturation (2,000 TPS on a system that saturates at $\sim 18,000$ TPS). To determine whether the +37% P95 overhead of $C = 1$ remains stable under higher utilization, we sweep load from 50% to 100% of saturation with epoch size $E = 1,000$, testing $C \in \{0, 1, 10, 50, 100, 1,000\}$ with finer load resolution at 75% and 80% (210 experiments; see Section 5.4).

Compaction overhead rises sharply with load (Table A.1). For $C = 1$, P95 grows from 230 ms at 50% load to 8,711 ms at 70%, and 55,489 ms at 85%. Its peak overhead is $69\times$ baseline at 80%. The threshold lies between 50% and 70% load. $C = 10$ stays within $2\times$ baseline through 75% but reaches $7.3\times$ at 85%.

For the moderate intervals ($C \geq 10$), the finer resolution at 75% and 80% locates each interval's own threshold. $C = 50$ stays within roughly $2\times$ of baseline at 75% (+17%) and 80% (+107%) but reaches $3.4\times$ at 85%. $C = 100$ is the most stable: it stays within about $2\times$ of baseline all the way through 85% (+57%; the worst point is +104% at 70%), with no throughput penalty at any load level.

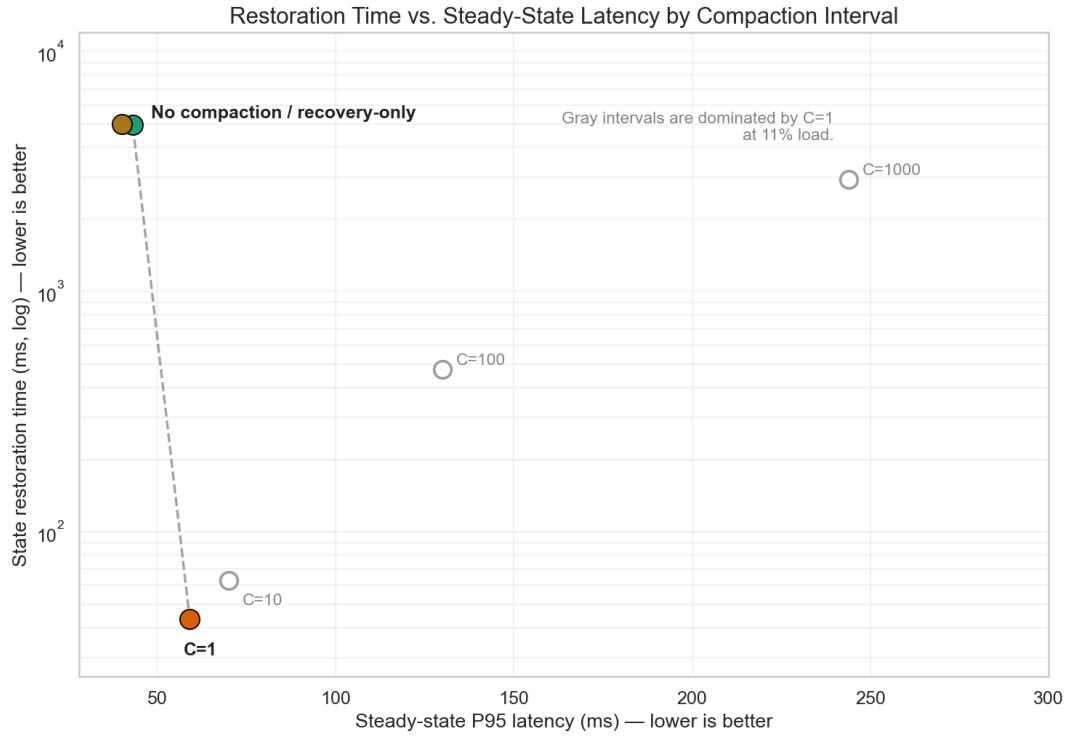


Figure 6.8: Restoration time (log scale) versus steady-state P95 latency. Colored points form the Pareto frontier; at 11% load, gray points are dominated by $C = 1$.

Table 6.11: P95 latency (ms) by load and compaction interval ($E = 1,000$; 250 s; 5 repetitions). $C = 0$ is the reference. No interval is best at every load: $C = 1$ degrades sharply above 50%, while $C = 100$ stays within about $2\times$ of the reference through 85%. The shaded columns mark this comparison.

Load	$C = 0$	$C = 1$	$C = 10$	$C = 50$	$C = 100$	$C = 1,000$	TPS ($C = 0$)
50%	106	230	164	169	183	360	8,855
70%	239	8,711	559	473	488	1,594	12,465
75%	478	25,344	683	561	605	1,896	13,242
80%	658	45,684	2,787	1,362	952	923	14,282
85%	1,721	55,489	12,539	5,806	2,702	2,826	15,255
95%	33,998	88,233	46,825	39,582	40,120	32,137	15,654
100%	52,109	113,624	65,554	59,750	52,583	52,596	15,703

Near saturation, small increases in storage work can create large queues. $C = 1$ adds storage traffic every epoch, while moderate intervals amortize compaction over more epochs. At $C = 100$, compaction runs about once per 200 epochs in this suite, so its I/O cost is small relative to transaction processing. Increasing C further reduces how often compaction runs but increases the number of files handled per event.

$C = 1,000$ shows a non-monotonic pattern. At 50–75% load, its median P95 is $3.4\text{--}6.7\times$ baseline and exceeds that of $C = 100$. At 80–85%, it approaches $C = 100$ but remains 40–64% above baseline. Only at 95–100% does its median P95 converge to baseline. A plausible explanation is that infrequent compactions merge larger batches and create isolated I/O bursts while the system still has spare capacity. Near saturation, baseline queueing dominates median P95, making the additional compaction cost difficult to distinguish. The 80% repetitions range from 630 to 3,303 ms. Because this interval permits only a few compactions per run and their timing was not instrumented, the measurements do not establish whether event timing caused the variation.

Figure 6.9 shows P95 latency on a log scale in the left panel and the overhead ratio relative to $C = 0$ in the right panel.

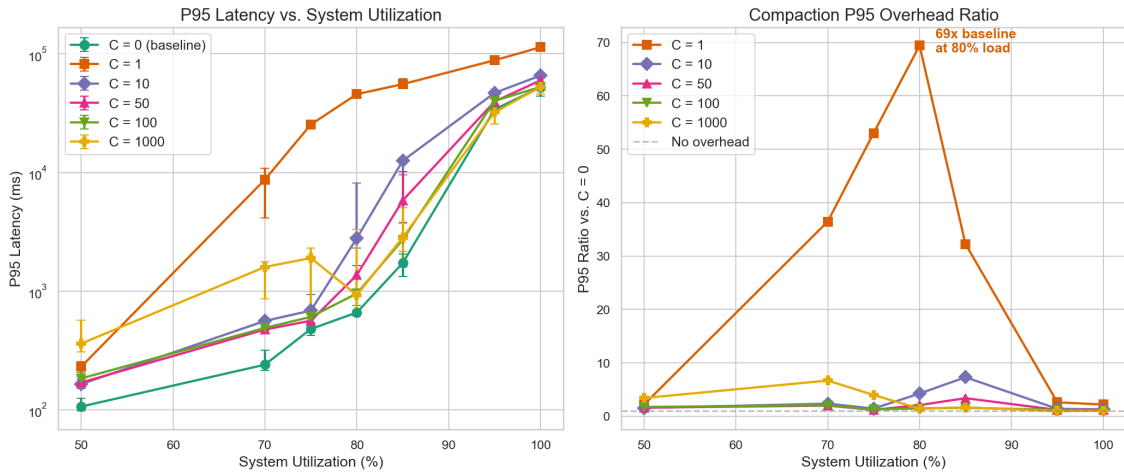


Figure 6.9: P95 latency versus utilization for all six intervals. Left: absolute P95 on a log scale. Right: ratio to $C = 0$.

Throughput is largely unaffected by compaction for $C \geq 10$: all such intervals stay within 3% of the $C = 0$ baseline through 85% load ($C = 10$ dips to about -6% at 95–100%). The sole exception is $C = 1$, which loses 15–22% of throughput at loads $\geq 80\%$ (2–9% at 70–75%) due to sustained I/O contention with stable storage. This confirms that the compaction overhead at high utilization is primarily a *latency* phenomenon for moderate intervals, not a throughput bottleneck.

6.2.6. Continuous vs. Recovery-Only Compaction (RQ2d)

Table 6.12 compares $C = 1$ with recovery-only compaction. Recovery-only defers all merging to a background thread that runs *after* state restoration, so the first recovery follows the baseline code path: fetch all accumulated delta files, merge them, and return the reconstructed state. The gap is two orders of magnitude (4,998 ms vs. 43 ms), and the repetition ranges of the two configurations do not overlap.

Table 6.12: Continuous ($C = 1$) vs. recovery-only compaction. Continuous makes the first restoration 115 $\times$ faster, while recovery-only avoids steady-state latency overhead. In the measured restart window, its background merge does not finish in time to speed the second restoration. **Bold** marks each mode’s measured advantage.

Metric	Best Continuous ($C = 1$)	Recovery-Only
First restoration time (ms)	43	4,998 ($\approx$ baseline)
Files fetched at first restore	7	3,040
Background compaction	No	Yes (after restore)
Second restoration time (ms)	22	3,649
Steady-state latency impact	+37% (P95: 59 ms)	$\approx 0\%$ (P95: 40 ms)

The background merge and sequencer garbage collection do not block transaction processing; the worker resumes serving requests immediately.

However, background compaction does *not* finish before the second recovery phase (rebalance). Recovery 2 for recovery-only processes 1,522 files (median) and takes 3,649 ms, within 23% of the baseline (2,967 ms, 1,497 files). The container restart interval (~ 1 s for Docker restart plus ~ 8 s for redistribution) is simply too short for background compaction of $\sim 3,000$ files to complete.

The choice depends on the restart window. In the measured ~ 9 s window, recovery-only adds no normal-operation cost but never finishes soon enough to speed the next restore. $C = 1$ raises P95 by 37%

but makes restoration nearly immediate. Longer restart windows could let recovery-only compaction finish.

6.2.7. Two-Phase Recovery and End-to-End Timeline

Each failure event triggers two sequential recovery phases: (1) *redistribution recovery*, where survivors restore the failed worker’s partitions, and (2) *rebalance recovery*, where all workers restore state after the failed worker rejoins. Table 6.13 shows the timing of both phases.

Table 6.13: Two-phase recovery breakdown (median values in milliseconds, 1 failure). R1 = redistribution recovery (survivors restore the failed worker’s partitions); R2 = rebalance recovery (all workers restore after the failed worker rejoins). The R1/R2 ordering varies by interval and the split is approximate for $C \leq 10$ (see text); both shrink by two orders of magnitude under aggressive compaction. **Bold** marks the fastest total. Medians are computed per column, so the R1 and R2 medians need not sum exactly to the total median.

Config.	R1 (ms)	R1 Files	R2 (ms)	R2 Files	Total (ms)
Baseline	4,971	2,988	2,967	1,497	8,151
$C = 1$	43	7	22	5	66
$C = 10$	63	27	43	17	98
$C = 100$	474	257	243	136	717
$C = 1,000$	2,941	1,726	1,578	865	4,494
Recovery-only	4,998	3,040	3,649	1,522	8,705

Recovery 2 is faster than Recovery 1 for the baseline and $C \geq 100$ configurations. For $C \leq 10$, both phases finish within tens of milliseconds, but their exact split is approximate. The restarted worker logs only its rebalance-phase cold restore, which the analysis pipeline attributes to Recovery 1. Rebalancing also spreads partitions across all 4 workers instead of 3 survivors, one of which carries 2 partitions, and thus reduces per-worker I/O. With recovery-only compaction, both phases remain close to baseline because background compaction does not finish before rebalancing.

The internal Recovery 1 and Recovery 2 timings cover state restoration, not the complete failure timeline. Figure 6.10 therefore compares their combined duration with the directly measured interval from fault injection until the cluster resumes after rebalancing. The remaining time includes failure handling, redistribution waits, container restart and readiness, messaging, and synchronization; this experiment does not isolate those contributions further.

With $C = 1$, the median measured timeline is 12.1 s, while both state-restoration phases together take 66 ms. At $C = 100$, the corresponding values are 12.0 s and 717 ms. Without compaction, the total rises to 20.2 s, of which 8.2 s is state restoration. Compaction therefore materially shortens the measured end-to-end timeline, but once restoration is below a second, other recovery work dominates.

6.2.8. Discussion

The linear recovery model and cloud storage implications. The linear fit ($R^2 = 0.998$, 1.67 ms/file) characterizes combined listing and fetching overhead in the local MinIO setup. AWS documents 100–200 ms latencies for small S3 objects⁴. If recovery performs thousands of comparable object operations sequentially, restoration could take minutes rather than seconds. Cloud restoration cost depends on listing behavior, request parallelism, caching, and network placement.

Writes are less concerning. Each checkpoint issues two PUTs per owned partition. Even the stress cadence produces only a few hundred requests per second across eight prefixes, about two orders of magnitude below S3’s per-prefix limit. At the recommended ~10-second checkpoint cadence, the cluster

⁴<https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html>

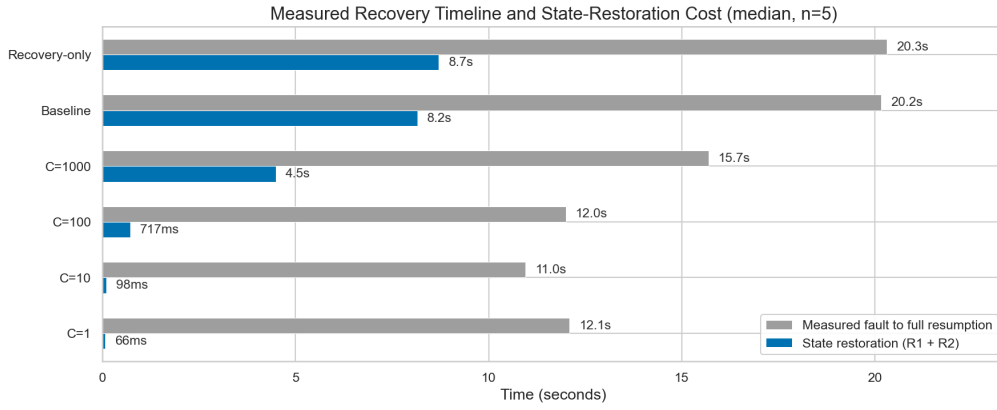


Figure 6.10: Measured time from fault injection to full cluster resumption compared with the combined Recovery 1 and Recovery 2 state-restoration time (median of 5 repetitions). The difference includes failure handling, redistribution, restart, readiness, messaging, and synchronization. Survivors continue at reduced capacity during part of this interval.

issues fewer than one PUT per second.

Practical recommendations. The best compaction interval depends on utilization; the low-load recovery results (Table 6.10) and load-dependent overhead sweep (Table 6.11) materially change the recommendation.

At 11% utilization, $C = 1$ is the best-tested state-restoration choice: its state restoration time is 99.1% lower (4,971 ms $\rightarrow$ 43 ms) and its latency overhead is lowest among compaction-enabled configurations (P95 +37%); at 50%, that overhead reaches +117%, where moderate intervals are cheaper (Table 6.11).

At moderate-to-high utilization (70–85%), $C = 1$ causes severe latency degradation (32 $\times$ baseline P95 at 85%) and $C = 10$ also degrades sharply (7.3 $\times$ at 85%). Of the tested settings, $C = 100$ offers the best and most stable trade-off across this range: it stays within 2 $\times$ of baseline latency at 85% (+57%) with no throughput penalty while reducing state restoration time by 90.5% (474 ms vs. 4,971 ms). Its 717 ms combined state-restoration time is about 6% of the measured 12.0 s fault-to-resumption timeline (Figure 6.10). At low load, its steady-state P95 is higher (130 ms vs. 59 ms at 11%; Table 6.10) because each event merges 100 checkpoint deltas; the overhead narrows as baseline P95 rises.

For latency-sensitive deployments where state restoration time is secondary, recovery-only compaction provides baseline-equivalent performance at the cost of slow state restoration. For this workload at moderate-to-high tested utilization, $C = 100$ offers the best-tested trade-off between state restoration time and steady-state overhead.

These intervals reflect this suite’s stress cadence, calibrated at this section’s start. At the $\sim$ 10-second checkpoint spacing appropriate at saturation (Section 6.1.4), $C = 1$ means one compaction every $\sim$ 10 s, and the baseline’s $\sim$ 1,500 files per partition ($\sim$ 5,800 across four partitions; Table 6.8) correspond to roughly four hours of uptime. Read the results as costs per unit of file accumulation, not wall-clock rates.

6.2.9. Consistency Verification

All 240 experiments (30 from the compaction recovery suite and 210 from the load-dependent overhead suite) passed consistency verification: every run confirmed exactly-once output with zero wrong values. Under the tested workload and storage assumptions, epoch-delta compaction produced no observed transactional correctness violations across the tested intervals, strategies, and utilization levels.

6.3. Rebalance Recovery Results (RQ3a–RQ3b)

6.3.1. Rebalancing Overhead (RQ3a)

Table 6.14 shows the coordinator-internal timing breakdown of the rebalance process. The total rebalance overhead, defined as the stop-the-world window during which no transactions are processed, is the sum of all phases. Values are the median of the 5 rebalance-condition repetitions with 4 workers and 1 failure (Phase 1’s matched degraded runs provide the no-rebalance baseline); the cluster runs at saturation load up to the moment of failure, after which the workload generator is stopped, so the breakdown times the recovery cycle itself ($C = 1$ compaction with cutoff margin $B = 100$).

Table 6.14: Rebalance timing breakdown (1-failure, $w = 4$, $C = 1$, $B = 100$, median of 5 repetitions). Phase values are per-phase medians with [min–max] across repetitions; because the median operator is not additive, phase medians do not sum exactly to the total median. Pausing workers (46%) and restoring the worker that gains partitions (37%) account for most of the ~827 ms window (**bold**); redistribution is negligible (<1 ms).

Phase	Time (ms)	[Min–Max] (ms)	Phase/Total
Pause workers	378	[229–592]	45.7%
Redistribute operators	<1	[<1]	<0.1%
Query MinIO for epoch	49	[48–51]	5.9%
Send InitRecovery	2	[2–2]	0.2%
Worker state restore	305	[289–375]	36.9%
Finalize recovery	101	[101–102]	12.2%
Total overhead	827	[686–1,118]	100%

Two phases dominate rebalancing: the worker pause (378 ms median, 46%) and state restore (305 ms, 37%). Pause time varies from 229 to 592 ms because the coordinator must wait for acknowledgments and residual in-flight work. During restore, the returning worker fetches about 100 files from the recent uncompact window created by the $B = 100$ cutoff margin. Listing, fetching, and merging take about 160 ms; messaging and setup account for the rest. The other three workers receive only a routing update.

Operator redistribution is an in-memory operation that reassigns partition ownership among surviving workers, completing in under 1 ms. The finalize phase (101 ms) is dominated by a fixed heartbeat-interval wait, and the MinIO epoch query (49 ms) scans the checkpoint data files to determine the recovery target epoch.

Rebalancing stops processing for about 827 ms. The $B = 100$ cutoff margin leaves about 100 files uncompact in these runs, which the returning worker must fetch. A smaller margin could reduce the ~160 ms fetch-and-merge work toward the 43 ms single-batch case, but pause time would still dominate.

From fault injection to resumed processing, the median is 11,772 ms without rebalance and 11,823 ms with it. Detection and restart dominate both. Their ranges overlap ([10,778–11,841] vs. [11,686–11,879] ms), so the 51 ms median difference does not isolate the rebalance cost. It only shows that rebalancing does not materially extend the end-to-end timeline.

6.3.2. Latency and Capacity Benefit (RQ3b)

Rebalancing pauses the cluster for 827 ms. Phase 2 asks whether the balanced assignment later repays that cost. It compares a *degraded* condition, where survivors keep extra partitions, with a *rebalance* condition across four loads. Table 6.15 reports matched throughput and end-to-end P95 during the observation window (median of 5 repetitions).

Table 6.15: Median per-second matched post-failure TPS and end-to-end P95 latency ($w = 4$, $C = 1$, $B = 100$; median of 5 repetitions). At 40%, rebalancing halves P95 with no material throughput change. At higher loads, the degraded condition has higher throughput and lower P95. **Bold** marks these meaningful differences.

Load	Condition	TPS	P95 (ms)
40%	Degraded	7,174	288
40%	Rebalance	7,176	146
50%	Degraded	8,411	36,400
50%	Rebalance	5,723	56,400
70%	Degraded	9,000	229,800
70%	Rebalance	5,412	291,500
85%	Degraded	8,696	281,700
85%	Rebalance	6,311	344,500

Observation at 40% load. At 40% load (7,200 TPS total input), both conditions deliver nearly identical throughput: 7,174 TPS degraded vs. 7,176 TPS rebalanced. P95 latency, however, diverges: 288 ms degraded vs. 146 ms rebalanced, a $2.0\times$ reduction. Even at this low load, the partition imbalance in the degraded configuration creates queuing at the ARIA epoch barrier: one survivor carries two partitions, taking longer to complete each epoch and forcing the other two to wait. Rebalancing eliminates this bottleneck. The effect is visible only in latency, not throughput, because the system has sufficient aggregate capacity to sustain the input rate.

Observation at 50% load. At 50% load (9,000 TPS total input), the cold-worker penalty emerges. Degraded achieves 8,411 TPS (93% of input) while rebalanced drops to 5,723 TPS (64%). P95 latency is 36.4 s degraded vs. 56.4 s rebalanced. The doubly loaded survivor processes epochs slower, but the ARIA barrier forces all workers to match the slowest. In the rebalanced configuration, the freshly restored worker is slower still, and the ARIA barrier forces the three warm workers to wait for it.

We did not isolate the cause. Possibilities include connection setup, Kafka seeking, and scheduling overhead on the restored worker. The state is only ~ 80 KB, so data-cache warming is unlikely to dominate. We use “cold worker” only as a label for the observed slowdown. Regardless of cause, the barriers make every worker wait for the slowest one.

Observations at 70% and 85% load. At 70% load (12,600 TPS total input) and 85% load (15,300 TPS), both conditions are deeply overloaded and the pattern from 50% persists. The cold worker’s epoch processing time dominates the ARIA barrier. Although one survivor carries twice the normal partition load in the degraded condition, it still outperforms the rebalanced condition on throughput and latency. The throughput advantage peaks at 70% (+66%: 9,000 vs. 5,412 TPS) and remains large at 85% (+38%: 8,696 vs. 6,311 TPS). P95 follows the same ordering: 230 s vs. 292 s at 70%, and 282 s vs. 344 s at 85%. Because these values mainly reflect a growing queue, sustained throughput provides the more reliable comparison.

Observation-window qualification. Phase 2 observes each post-recovery condition for nominally 90 seconds. At loads $\geq 50\%$, the systems are overloaded and the request cohort extends beyond that window. The reported P95 therefore measures queue growth more than ordinary service latency. We use sustained throughput for comparison in this regime. The window is too short to establish whether the cold-worker penalty eventually disappears.

Break-even analysis. Within this 90-second observation window, no break-even point was observed at any tested load level. At 40%, both conditions sustain the input (steady-state throughput ratios 1.00 rebalanced vs. 0.996 degraded, below the 2% reporting criterion of Chapter 5), so throughput is identical and rebalancing wins on latency. At 50–85%, degraded outperforms rebalanced on throughput, so the cumulative throughput curves never cross during the window. We could not determine whether the penalty is transient (the restored worker speeds up given enough time) or persistent (the ARIA barrier permanently limits throughput when one worker is slower).

6.3.3. Consistency Verification

All 40 Phase 2 benefit experiments ($f = 1$; 5 reps $\times$ 4 loads $\times$ 2 conditions) passed consistency verification with zero wrong values. Thus, under this failure model and workload, operator redistribution, state restore from stable storage, and epoch-aligned resumption produced no observed consistency violations, including at the severely overloaded 70% and 85% conditions.

6.3.4. Limitation: High-Ratio Failures

Our experiments evaluate single-worker failures ($f = 1$). The current rebalance implementation does not support high-ratio failures ($f \geq \lceil w/2 \rceil$). For example, killing 3 of 4 workers leaves one survivor to absorb the full partition load, which exceeds the coordinator’s recovery timeout. Supporting such failures would require a longer timeout or staged recovery that migrates partitions incrementally; we leave both options to future work.

6.4. Summary

The evaluation shows that uncoordinated checkpointing preserves SRYX’s steady-state performance under the tested conditions (RQ1). Epoch-delta compaction cuts state restoration time by up to two orders of magnitude. For this workload, $C = 100$ gives the best tested balance and reduces state restoration time by $\sim 90\%$ (RQ2). Partition rebalancing improves tail latency at low load but incurs a cold-worker penalty at higher utilization (RQ3).

Conclusion and Future Work

7.1. Summary

This thesis studied whether moving checkpoint persistence from the coordinator to the workers preserves steady-state performance and practical recovery. Stryx’s lockstep epoch boundaries let workers persist independently, while recovery selects the latest epoch available for every partition.

The evaluation addresses the three resulting concerns identified in Section 1.2. Uncoordinated checkpointing preserves steady-state performance under the tested conditions. Compaction controls delta-file growth and substantially reduces state restoration time, although aggressive compaction increases foreground latency. Rebalancing is workload-dependent: it improves latency at the lowest tested load but lowers throughput at higher loads during the 90-second observation window. Chapter 6 provides the supporting measurements.

7.2. Answers to the Research Questions

7.2.1. RQ1: Steady-State Performance

RQ1. Under the tested single-node conditions, moving checkpoint persistence from the coordinator to the workers produces no systematic steady-state performance difference. The coordinator remains necessary for execution barriers; this conclusion applies only to its checkpoint write (Section 6.1).

RQ1a. Throughput increases with worker count, but sublinearly. Shared execution barriers and the single-node environment limit scaling, so adding workers alone does not produce a proportional capacity increase (Section 6.1.2).

RQ1b. Across the six worker counts, an epoch size of 100 provides the lowest aggregate median tail latency while preserving most throughput. The result varies at the smallest scale, where size 250 is lowest; sizes from 100 to 250 therefore form the best tested range rather than a universal prescription (Section 6.1.3).

RQ1c. The checkpointing protocol introduces no systematic throughput or latency cost at sustainable load. Very frequent checkpointing can increase latency, while moderate and sparse cadences perform similarly (Section 6.1.4).

RQ1d. Contention, rather than checkpointing mode, sets the capacity boundary. Adding workers does not overcome hot-key serialization and can worsen scaling relative to uniform access. At $z = 0.99$, both modes overload before validation completes, so the exactly-once evidence is limited to the tested boundary $z \leq 0.75$ (Section 6.1.5).

7.2.2. RQ2: Compaction and Recovery

RQ2. Epoch-delta compaction reduces state restoration time by limiting the number of files read from stable storage. The trade-off is not monotonic. Compacting every checkpoint favors recovery speed but becomes disruptive as load rises, while very sparse compaction produces higher and more variable latency at lower loads. Moderate intervals best protect foreground latency across the tested load range (Section 6.2).

RQ2a. Reducing the delta-file count reduces state restoration time almost linearly, making file count the main practical recovery lever in this design (Section 6.2.2).

RQ2b. State restoration is I/O-bound in the tested environment. Listing and fetching files dominate the measured time, while merging contributes little (Section 6.2.3).

RQ2c. No interval is best at every load. Compacting every 100 checkpoints ($C = 100$) provides the best tested balance between restoration time and foreground latency. Compacting every checkpoint restores faster but becomes too disruptive as load rises (Sections 6.2.4 and 6.2.5).

RQ2d. Recovery-only compaction does not provide the same restoration benefit as continuous compaction. It avoids most steady-state overhead but leaves the expensive file reduction work on the recovery path (Section 6.2.6).

7.2.3. RQ3: Rebalancing After Recovery

RQ3. Rebalancing restores the pre-failure partition distribution, but not necessarily its performance. Because every barrier waits for the newly restored worker, the benefit depends on whether the cluster can absorb its warm-up cost (Section 6.3).

RQ3a. Rebalancing adds an 827 ms stop-the-world interval in the tested setup. Pausing workers and restoring state dominate this cost; metadata redistribution is negligible (Section 6.3.1).

RQ3b. Rebalancing improves tail latency without reducing throughput at 40% utilization. At loads of 50% and above, it lowers throughput during the 90-second observation window, and no break-even point appears. The evidence cannot determine whether this penalty is temporary or persistent (Section 6.3.2).

Together, these findings show that checkpoint persistence can be decoupled from the coordinator, while recovery cost and post-failure rebalancing remain workload-dependent.

7.3. Limitations

Scope of evidence. The evaluation covers one benchmark and a single-node deployment. The scalability study varies the cluster from 2 to 12 workers, whereas the compaction and rebalance studies use a fixed 4-worker cluster; the failure experiments primarily inject one worker failure. The findings therefore describe the tested workload and environment rather than universal performance or recovery guarantees.

Measurement boundaries. The study verifies recovered state and characterizes run-to-run variation, but it does not fully measure realized work loss, long-term rebalance behavior, or statistical equivalence between close configurations.

7.4. Future Work

7.4.1. Adaptive Compaction Scheduling

The best tested compaction interval changes with load: aggressive compaction accelerates restoration but can sharply increase foreground latency. Future work should evaluate whether a policy based on utilization, file count, and state growth can meet a restoration-time target while limiting latency overhead and preserving the recovery line (Section 6.2.5).

7.4.2. Diagnosing and Reducing the Cold-Worker Penalty

The current experiments do not isolate why a restored worker initially limits epoch progress. Future work should separate the effects of state restoration, Kafka seeking, connection setup, and scheduling. It should then evaluate whether worker pre-warming or live peer-to-peer state transfer shortens this period and allows rebalancing to recover its initial cost at higher loads.

7.4.3. Long-Duration Rebalance Evaluation

The 90-second post-recovery window cannot distinguish a transient warm-up cost from a persistent penalty. Longer runs should measure how long the restored worker takes to reach stable throughput and whether the cumulative benefit of rebalancing eventually exceeds its initial cost. This would establish when, if ever, rebalancing breaks even at each load.

7.4.4. High-Ratio Failure Recovery

Failures of at least half the workers can overload the survivors until recovery times out (Section 6.3.4). Future work should compare longer timeouts with staged redistribution. This would identify the supported failure ratios and show how recovery cost scales with the number of failures.

7.4.5. Validation Under Extreme Contention

At $z = 0.99$, validation ends with transactions still queued, so the results cannot distinguish overload from a correctness problem. The workload should be rerun at a skew-specific sustainable rate and allowed to drain before final-state verification. Instrumentation should record submitted, committed, queued, and verified transactions. If mismatches remain, the next step is to trace the shared ARIA fallback path and reproduce the behavior on an unmodified Sryx baseline. This would test exactly-once processing under extreme contention (Section 6.1.5).

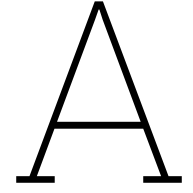
7.4.6. Distributed Deployment

All experiments run on one machine with loopback networking and local object storage, which understates barrier and storage latency. Multi-node experiments with remote object storage should measure execution barriers, checkpoint persistence, and restoration separately. They would show whether uncoordinated checkpointing retains its relative steady-state performance, whether file count remains the main restoration cost, and whether the recommended compaction interval changes in a distributed deployment.

7.5. Closing Remarks

This thesis began with a narrow question: must checkpoint persistence pass through the execution coordinator? For Stryx, the answer is no. Epoch barriers already establish consistent boundaries, so the coordinator does not need to establish consistency again when a checkpoint is written. Workers can persist their partition state independently while the execution protocol continues to coordinate transaction order.

This separation does not remove coordination or persistence failures. A worker that cannot write can still hold back the latest common recovery point, accumulated files make restoration expensive, and a restored worker can slow later epochs when it rejoins. Instead, the design narrows the coordinator's role to decisions that must be shared and leaves storage work with the workers that own the state. The broader lesson is to first ask whether the execution model already provides a consistent cut. When it does, an additional central checkpoint step may add a dependency without providing a stronger recovery guarantee.



Supplementary Results

A.1. Compaction Overhead as Percentages

Table A.1 restates the absolute P95 measurements of Table 6.11 relative to the no-compaction baseline.

Table A.1: P95 overhead relative to $C = 0$. $\uparrow$ marks overhead of at least 100%; $\downarrow$ marks a reduction; $\approx$ marks a change within 5%.

Load	$C = 1$	$C = 10$	$C = 50$	$C = 100$	$C = 1,000$
50%	$\uparrow +117\%$	+55%	+59%	+73%	$\uparrow +240\%$
70%	$\uparrow +3,545\%$	$\uparrow +134\%$	+98%	$\uparrow +104\%$	$\uparrow +567\%$
75%	$\uparrow +5,202\%$	+43%	+17%	+27%	$\uparrow +297\%$
80%	$\uparrow +6,843\%$	$\uparrow +324\%$	$\uparrow +107\%$	+45%	+40%
85%	$\uparrow +3,124\%$	$\uparrow +629\%$	$\uparrow +237\%$	+57%	+64%
95%	$\uparrow +160\%$	+38%	+16%	+18%	$\downarrow -5\%$
100%	$\uparrow +118\%$	+26%	+15%	$\approx +1\%$	$\approx +1\%$

B

Implementation File Map

Table B.1: Key implementation files for each contribution.

Contribution	Key Files
Checkpoint strategies	worker/fault_tolerance/checkpoint_strategies/, worker/.../aria_uncoordinated.py
Per-partition seq. files	worker/fault_tolerance/uncoordinated_snapshots.py
Epoch-delta compaction	worker/fault_tolerance/compaction.py
Recovery process	coordinator/uncoordinated_coordinator.py
Partition rebalancing	coordinator/uncoordinated_coordinator.py, coordinator/coordinator_service.py
Persistence-fault demonstration	demo/demo-coordinator-stall/ (recorded video included)

References

- [1] Daniel J Abadi and Jose M Faleiro. “An overview of deterministic database systems”. In: *Communications of the ACM* 61.9 (2018), pp. 78–88.
- [2] Amazon Web Services. *Amazon S3 now delivers strong read-after-write consistency*. <https://aws.amazon.com/about-aws/whats-new/2020/12/amazon-s3-now-delivers-strong-read-after-write-consistency-automatically-for-all-applications/>. 2020. (Visited on 03/28/2026).
- [3] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the Spring Joint Computer Conference (AFIPS '67)*. 1967, pp. 483–485.
- [4] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. “Netherite: Efficient execution of serverless workflows”. In: *Proceedings of the VLDB Endowment* 15.8 (2022), pp. 1591–1604.
- [5] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. “State management in Apache Flink: Consistent stateful distributed stream processing”. In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1718–1729.
- [6] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. “Lightweight asynchronous snapshots for distributed dataflows”. In: *arXiv preprint arXiv:1506.08603* (2015).
- [7] K Mani Chandy and Leslie Lamport. “Distributed snapshots: Determining global states of distributed systems”. In: *ACM Transactions on Computer Systems* 3.1 (1985), pp. 63–75.
- [8] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. “Rhino: Efficient management of very large distributed state for stream processing engines”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2471–2486.
- [9] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. “YCSB+T: Benchmarking web-scale transactional databases”. In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. 2014, pp. 223–230.
- [10] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. “RocksDB: Evolution of development priorities in a key-value store serving large-scale applications”. In: *ACM Transactions on Storage* 17.4 (2021), pp. 1–32.
- [11] E N Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. “A survey of rollback-recovery protocols in message-passing systems”. In: *ACM Computing Surveys* 34.3 (2002), pp. 375–408.
- [12] Avrielia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. “Dhalion: Self-regulating stream processing in Heron”. In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1825–1836.
- [13] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. “A survey on the evolution of stream processing systems”. In: *The VLDB Journal* 33.2 (2024), pp. 507–541.

- [14] Yu Gan, Yanqi Zhang, Dailun Cheng, et al. "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 3–18.
- [15] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. "Meces: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 539–556.
- [16] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. "Distributed transactions on serverless stateful functions". In: *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. 2021, pp. 31–42.
- [17] Torsten Hoefer and Roberto Belli. "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, 2015, 73:1–73:12.
- [18] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. "Megaphone: Latency-conscious state migration for distributed streaming dataflows". In: *Proceedings of the VLDB Endowment* 12.9 (2019), pp. 1002–1015.
- [19] Zhipeng Jia and Emmett Witchel. "Boki: Stateful serverless computing with shared logs". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 691–707.
- [20] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 783–798.
- [21] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB Workshop*. 2011, pp. 1–7.
- [22] Avinash Lakshman and Prashant Malik. "Cassandra: A decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.
- [23] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401.
- [24] Scott T Leutenegger and Daniel Dias. "A modeling study of the TPC-C benchmark". In: *ACM SIGMOD Record* 22.2 (1993), pp. 22–31.
- [25] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. "Aria: A fast and practical deterministic OLTP database". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2047–2060.
- [26] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. "Mencius: Building efficient replicated state machines for WANs". In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. 2008, pp. 183–198.
- [27] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.
- [28] Kyriakos Psarakis, George Christodoulou, George Siachamis, Marios Fragkoulis, and Asterios Katsifodimos. "Styx: Transactional stateful functions on streaming dataflows". In: *Proceedings of the ACM on Management of Data* 3.3, 226 (2025).

- [29] Sheng Qi, Xuanzhe Liu, and Xin Jin. “Halfmoon: Log-optimal fault-tolerant stateful serverless computing”. In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023, pp. 314–330.
- [30] Kun Ren, Thaddeus Diamond, Daniel J Abadi, and Alexander Thomson. “Low-overhead asynchronous checkpointing in main-memory database systems”. In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 2016, pp. 1539–1551.
- [31] George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, and Asterios Katsifodimos. “CheckMate: Evaluating checkpointing protocols for streaming dataflows”. In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2024, pp. 4030–4043.
- [32] Pedro Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. “Clonos: Consistent causal recovery for highly-available streaming dataflows”. In: *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 2021, pp. 1637–1650.
- [33] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. “Cloudburst: Stateful functions-as-a-service”. In: *Proceedings of the VLDB Endowment* 13.11 (2020), pp. 2438–2452.
- [34] Alexander Thomson and Daniel J Abadi. “The case for determinism in database systems”. In: *Proceedings of the VLDB Endowment* 3.1–2 (2010), pp. 70–80.
- [35] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel 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.
- [36] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. “Fault-tolerant and transactional stateful serverless workflows”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 1187–1204.
- [37] Shuhao Zhang, Juan Soto, and Volker Markl. “A survey on transactional stream processing”. In: *The VLDB Journal* 33.2 (2024), pp. 451–479.
- [38] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. “Starry: Multi-master transaction processing on semi-leader architecture”. In: *Proceedings of the VLDB Endowment* 16.1 (2022), pp. 77–89.
- [39] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. “Lotus: Scalable multi-partition transactions on single-threaded partitioned databases”. In: *Proceedings of the VLDB Endowment* 15.11 (2022), pp. 2939–2952.