Engineering Fractal: Runtime Design, Optimizations, and Evaluation for Fault-Tolerant Shell-Script Distribution

Ramiz Dundar
Brown University

Abstract

We present the runtime design and implementation of FRACTAL, a system for fault-tolerant distributed execution of unmodified POSIX shell scripts. FRACTAL distinguishes recoverable regions from side-effectful regions and augments distributed subgraphs with runtime support for progress tracking, exactly-once communication, selective replay, and dynamic output persistence. We describe the recovery workflow, the remote-pipe data path, the distributed file reader, and the executor-side optimizations that keep fault-tolerance overhead low on the common path. Evaluation on 4- and 30-node clusters shows that FRACTAL preserves the performance benefits of distributed shell execution while recovering 7.8–16.4× faster than Hadoop Streaming under worker faults.

1 Introduction

The Unix shell remains one of the most widely used environments for real-world data processing and systems work because it combines language-agnostic composition, lightweight pipelines, and dynamic control flow in a compact programming model [17, 20, 23, 25, 44, 50]. Recent systems such as PaSh, POSH, and DISH have shown that many shell scripts can be parallelized and distributed automatically without rewriting them into a new framework [28, 39, 43, 56]. Transparent recovery from worker faults is still missing. That omission matters precisely in the environments that motivate distributed shell execution: long-running scripts, large inputs, commodity clusters, and black-box commands whose internal state remains opaque to the runtime.

FRACTAL addresses this gap by pairing shell-aware partitioning with recovery-oriented runtime support. It separates recoverable computation from side-effectful regions, records progress at subgraph boundaries, and re-executes only the affected portions of a distributed execution after worker faults. This report concentrates on the mechanisms that make that behavior practical, including the recovery workflow, progress tracking, discovery, remote pipe communication, distributed file reading, executor control flow, and the optimizations that balance recovery speed against fault-free overhead.

Contributions. This report focuses on Fractal's runtime design and implementation. Its emphasis is on the execution engine, targeted optimizations, and the evaluation evidence that characterizes these mechanisms in practice. The broader design-space characterization, the reference-prototype view of Fractal, and the fault-injection tooling appear only where needed to keep the runtime story self-contained.

This report makes four contributions. It presents a runtime architecture for recovery in distributed shell execution, tying together progress tracking, discovery, remote pipe communication, and replay coordination. It explains how byte-accurate offset accounting, subgraph dependency tracking, and recovery-safe data movement make selective recovery feasible for black-box shell workloads. It surfaces the engineering tradeoffs behind dynamic persistence, buffered I/O, event-driven execution, and batched scheduling. It evaluates those runtime mechanisms on 4- and 30-node clusters with an emphasis on fault-free overhead, recovery behavior, and persistence choices, treating fault injection as experimental methodology rather than as a separate subsystem contribution.

The rest of the report moves from background to implementation and then to empirical results. Section 2 summarizes the shell-specific requirements that shape the runtime. Section 3 introduces the high-level workflow. Sections 4 and 5 cover the runtime design and its optimizations. Section 6 then describes the experimental setup and the main empirical results. Sections 7 to 9 close the report with context, caveats, and the broader implications of the runtime design.

2 Background and Design Space

This section summarizes the design-space characterization from the Fractal project that motivates the runtime mechanisms in the rest of the report. It outlines the shell-specific desiderata for fault-tolerant distribution, explains why established recovery patterns do not transfer directly, and then sketches Fractal's response at a high level.

Tab. 1: Comparison of fault-tolerance mechanisms across key shell desiderata. This table follows the Fractal paper's design-space comparison and explains the recurring D1 – D6 badges used throughout the report.

Desideratum	Checkpointing	Barrier-based	Lineage-based	FRACTAL
D1 Handles black-box state	No	Yes	No	Yes
D2 Ad-hoc pipe integrity	No	No	No	Yes
D3 Side-effect management	Part.	No	No	Yes
D4 Dynamism compatibility	Part.	No	No	Yes
D5 Recovery granularity	Coarse	Coarse	Fine	Fine
D6 No script modification	Part.	Part.	No	Yes

2.1 Desiderata

Shell scripts uniquely blend diverse commands, streaming pipelines, visible side effects, and dynamic expansion at runtime. That combination gives the shell much of its practical power, but it also creates a distinct set of requirements for any recovery design. Tab. 1 summarizes those requirements.

D1 Black-box state handling. Shell pipelines invoke external binaries such as `sort`, `grep`, and `unzip` whose internal state cannot be inspected or checkpointed by the runtime. These commands may buffer large amounts of data internally without exposing any API for partial snapshots or rollback. A fault-tolerance mechanism therefore has to recover progress without peeking inside opaque commands or demanding new hooks from them.

D2 Ad-hoc pipe streaming integrity. Shell commands communicate through unstructured byte streams over ad-hoc Unix pipes, with buffering and chunking behavior that varies by command. When a worker fails, the system has no built-in record of how many bytes a downstream consumer has already received. Recovery must therefore preserve exactly-once delivery across retries so that replay neither drops nor duplicates data.

D3 Side-effect management. Shell commands often perform non-idempotent external actions, such as appending to files, moving outputs into place, or issuing network calls. Simply rerunning a partially completed side-effectful command can duplicate visible effects or leave behind partial writes. A shell-oriented recovery design must either prevent those repeated effects or keep such commands outside the replay path.

D4 Dynamism compatibility. Shell scripts resolve control flow and command invocations at runtime through loops, conditionals, variable expansion, command substitution, and file-system reflection. Any practical design must handle these on-the-fly pipelines rather than assume that the full computation graph is known statically ahead of time.

D5 Fine recovery granularity. Shell workflows often chain many long-running commands, so coarse reruns waste substantial work after a partial failure. At the same time, command-level replay is hard because black-box commands hide internal state and because downstream consumers may already have observed ephemeral outputs. A robust mechanism therefore needs a recovery unit that is small enough to avoid large reruns but coarse enough to remain implementable.

D6 No script modification. Shell scripts are often legacy artifacts or incrementally maintained operational code. Forcing users to rewrite them into mapper/reducer wrappers, framework-native operators, or custom recovery APIs would sacrifice much of the shell's value. An ideal approach should preserve existing POSIX scripts unchanged while adding recovery support transparently [14, 15, 18, 21].

2.2 Existing Approaches

Existing recovery paradigms illuminate the design space, but none transfers cleanly to general shell execution [4, 5, 7, 10, 11, 22, 24, 30, 53, 59]. In practice, failing even one of D1 – D6 is enough to make a mechanism a poor fit for unmodified shell scripts.

Checkpointing. Checkpointing systems assume that the runtime can snapshot process or operator state through framework-visible hooks [4, 5, 7, 30, 53]. That assumption conflicts directly with D1: opaque shell commands do not expose internal state to checkpoint. It also conflicts with D2, because Unix pipes do not carry framework-managed offsets or barriers, and with D3, because shell-side file and network I/O occur outside transactional sink APIs. Checkpointing further struggles with D4 when new commands appear dynamically, and it tends to offer only coarse or expensive answers to D5. Finally, adapting shell workflows to checkpointing usually requires wrappers or rewrites that violate D6.

Barrier-based recovery. Barrier-based systems such as MapReduce and Hadoop Streaming tolerate faults by retrying larger task units against a mostly static execution graph [10, 11, 22, 30]. They satisfy D1 only partially, because they can rerun black-box tasks but cannot resume partially completed internal computation. They do not satisfy D2, since replaying raw

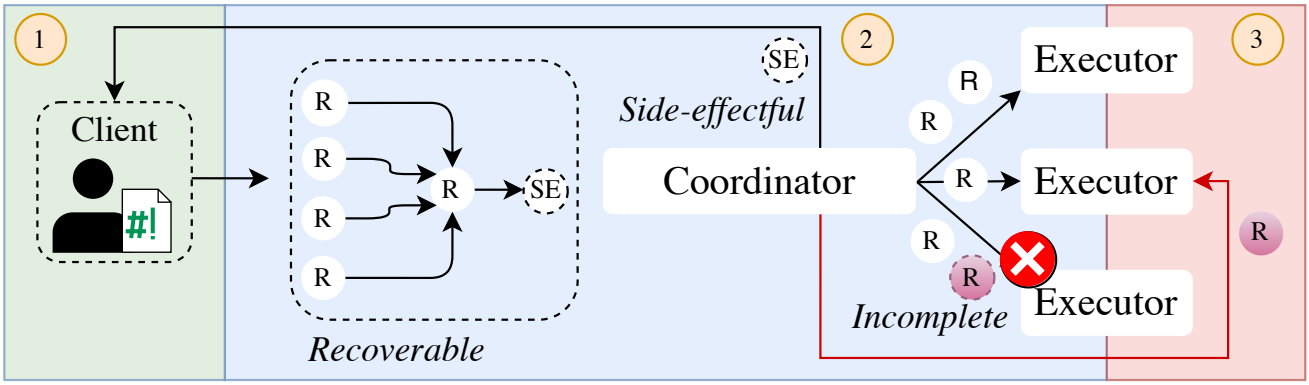

Fig. 1: Fractal's execution workflow. FRACTAL separates side-effectful regions from recoverable regions, executes recoverable subgraphs on workers while tracking progress and health, and reschedules only unfinished work after a failure.

byte streams across retries risks duplication unless the shell pipeline is rewritten around durable brokers or batch boundaries. They also fare poorly on D3 and D4: retries can re-issue side effects, and shell-generated dynamic branches do not map naturally onto a fixed task graph. Their task-oriented retry model is also too coarse for D5, and Hadoop Streaming still requires mapper/reducer wrappers that weaken D6.

Lineage-based replay. Lineage-based systems such as Dryad and Spark come closest on D5 because they can recompute only failed partitions and their dependencies [24, 59]. Even so, they remain a poor fit for the rest of the shell setting. They assume that operator behavior is reconstructible from recorded dependencies, which clashes with D1 for black-box commands whose internal progress is opaque. They do not naturally preserve D2 on ad-hoc byte streams, they rely on framework-managed sinks for D3, and they still expect explicit registration of operators and edges that conflicts with D4. Most importantly, they generally require scripts to be re-expressed inside the lineage framework, which violates D6.

2.3 Fractal's response

Fractal answers these constraints by treating a *subgraph* of the distributed shell execution as the unit of recovery. That choice avoids deep instrumentation of every individual command while remaining fine-grained enough to avoid large-scale reruns. At subgraph boundaries, the runtime injects lightweight support for progress tracking, replay-safe communication, and selective reuse of persisted outputs.

This model addresses the six desiderata directly. For D1, recovery relies only on subgraph inputs and outputs rather than internal command state. For D2, byte-level progress tracking and replay suppression preserve exactly-once downstream delivery. For D3, non-idempotent commands remain in the client-side `main` subgraph, while recoverable distributed subgraphs either perform pure transformations or write to isolated outputs. For D4, the compiler derives subgraphs from the shell program as executed, so runtime-dependent structure remains within the model. For D5, subgraph-level replay is precise enough to avoid coarse reruns without incurring the bookkeeping cost of per-command recovery. For D6, all of this support is injected automatically, so users still run unmodified POSIX shell scripts.

3 Fractal Overview

Overview. FRACTAL addresses shell fault tolerance by building on PaSh-JIT's data-flow representation of shell scripts [28]. At a high level, the coordinator identifies the script regions that must remain on the client node, partitions the rest into distributed subgraphs with different recovery semantics, and then augments the boundaries between those subgraphs with runtime support for replay-safe execution. The `main` subgraph contains the parts of the program that depend on authoritative shell state or may perform non-idempotent side effects. `regular` subgraphs do not include an aggregator vertex, whereas `merger` subgraphs include an aggregator vertex responsible for combining outputs from multiple upstream subgraphs.

Fig. 1 summarizes the resulting workflow. Before execution, FRACTAL instruments every inter-subgraph edge with runtime support for progress tracking and replay-safe communication. During execution, the runtime tracks subgraph placement, inter-subgraph dependencies, and byte-level communication progress. When the health monitor detects a worker fault, the coordinator computes the smallest replay plan consistent with those dependencies and redistributes only the unfinished work. The next section then unpacks that high-level picture by mapping the architecture's labeled stages to the concrete runtime subsystems.

4 Runtime Design and Implementation

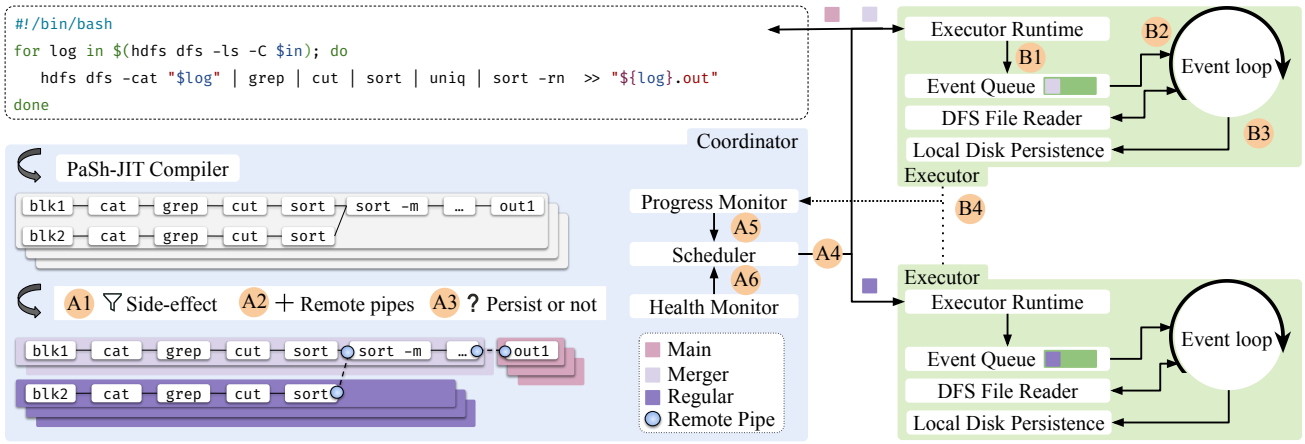

Fig. 2: Runtime-oriented view of the Fractal architecture. The compiler isolates the unsafe `main` region, instruments inter-subgraph edges, and hands prepared subgraphs to the scheduler. The runtime then relies on progress monitoring, discovery, remote pipe communication, distributed file reading, and executor-side control flow to support efficient recovery.

Fig. 2 shows the full path from compiled shell graph to worker execution. The runtime is designed to recover failed work without blindly replaying downstream computation that has already completed. That requirement drives the coordination path between recovery planning, progress tracking, inter-subgraph communication, and worker-side execution.

The orange architecture badges in Fig. 2 mark the coordinator-side path from compilation to recovery.

A1 Unsafe-main isolation. The compiler isolates side-effectful or shell-state-dependent regions into the client-side `main` subgraph and partitions the remaining distributed work into recoverable subgraphs, building on PaSh-JIT's command-level shell graph representation [28].

A2 Remote-pipe instrumentation. Inter-subgraph edges are replaced with remote pipe endpoints so the runtime can track byte-level progress and reconnect streams safely after a fault.

A3 Persistence decision. Before execution, the runtime decides whether each relevant edge should stream transiently or persist output locally, balancing fault-free overhead against recovery speed.

A4 Scheduling and dispatch. Prepared subgraphs are handed to the scheduler, which assigns them to workers and initiates the corresponding communication endpoints.

A5 Progress monitoring and discovery. The coordinator records completion events, dependency state, and endpoint metadata so that downstream consumers can locate writers and recovery can reuse completed work correctly.

A6 Health monitoring. The health monitor detects worker failures and triggers the recovery procedure that recomputes the minimal replay plan and reschedules only unfinished subgraphs.

4.1 Recovery workflow

When the A6 health monitor reports that a worker has failed, the coordinator rebuilds the execution plan in five steps. It identifies the incomplete subgraphs that were assigned to the failed node, uses the progress metadata to determine which upstream and downstream dependencies are affected, issues kill requests to work that no longer matches the new plan, updates the progress-monitor state, excludes outputs that can be reused from persistence, and finally reschedules the remaining subgraphs on healthy nodes. This recovery path closes the loop between A6 failure detection, A5 recorded progress, and A4 rescheduling.

The recovery action depends on the role of the failed subgraph. If a `regular` worker fails, FRACTAL can often restart only that subgraph and reconnect its downstream consumer at the correct byte offset. If a `merger` worker fails, the runtime creates a replacement merger and coordinates a mix of replayed and persisted upstream inputs. This asymmetry is important later in the evaluation: merger failures often cost more because they sit closer to shared aggregation paths, whereas regular-worker faults are more isolated.

4.2 Progress tracking and discovery

The A5 progress monitor is the runtime's source of truth for recovery-relevant metadata. It records subgraph placement, inter-subgraph dependencies, completion events, and, when persistence is enabled, the locations of reusable outputs. Each completed send or receive emits a compact 17-byte event consisting of an edge identifier and a direction flag. Those tiny records are enough for the coordinator to decide whether a downstream consumer has already received the bytes associated with a given upstream execution.

The A5 discovery service complements the progress monitor by solving a narrower problem: helping the endpoints of a distributed edge find each other after scheduling or rescheduling. Writers publish either a socket endpoint or a file-backed endpoint, and readers poll until matching metadata becomes available. Because discovery is logically separate from recovery planning, the runtime can keep the communication path simple while still allowing reconnected readers to resume once a replacement writer comes online.

4.3 Remote pipes and exactly-once delivery

Fractal's A2 remote pipe is the key runtime mechanism for inter-subgraph communication. It acts as a distributed replacement for a local Unix pipe and supports two modes. In transient mode, the writer exposes a socket endpoint for low-latency fault-free execution. In persistent mode, the writer exposes a file-backed endpoint so that already produced bytes can be reused after a fault.

The critical correctness property is exactly-once downstream delivery. Readers track how many bytes have already been forwarded downstream. If a writer is restarted after failure and begins replaying a stream, the reader discards duplicate bytes until it reaches the last known offset, then resumes forwarding new data. That offset accounting is what lets FRACTAL recover a failed worker without forcing every downstream consumer to start from scratch.

The implementation also has to detect stream completion without buffering the whole stream. Writers append a fixed 8-byte sentinel at end of stream. Readers retain a short look-ahead buffer so they can distinguish payload bytes from the sentinel while continuing to stream data promptly. This small detail is important because byte-accurate replay is only useful if the data path remains efficient on the common path.

4.4 Distributed file reader and executor runtime

Distributed shell execution also needs efficient input access. The distributed file reader serves HDFS-resident data splits directly to worker-side shell code so that execution can remain data-local when possible [46]. Instead of staging whole files through a client-side copy path, the runtime streams the relevant split from the local data node and feeds it into the worker's subgraph pipeline.

The executor runtime is the worker-side control loop that turns serialized subgraphs back into runnable shell fragments. It stages scripts and metadata in a temporary directory, launches pending work up to a configurable concurrency limit, reclaims completed processes, applies incoming kill requests, and records timing and diagnostic information. The design stays deliberately small: the more logic that moves into the runtime's critical path, the more likely recovery support is to erode the fault-free speedups that motivated distribution in the first place. Within Fig. 2, this subsection corresponds most directly to the worker-side consequences of A4 scheduling and the coordinator-side services that feed those executors.

4.5 Recovery boundaries and safe replay

Fractal still needs a clear answer to the question "what may be replayed safely?" It inherits a command-annotation catalogue from prior shell-distribution work [28, 39, 43, 56]. Commands that depend on local shell state or produce non-idempotent side effects remain in the client-side `main` subgraph; this is the concrete effect of A1 unsafe-main isolation. Commands that behave as recoverable data transformations can participate in the distributed execution plan, which gives the runtime a principled boundary between work that may be re-executed and work that must remain sequential and authoritative on the client side.

5 Optimizations and Engineering Tradeoffs

The runtime mechanisms in Section 4 are necessary for correctness, but they would not be useful in practice if they imposed large overheads on the common fault-free path. FRACTAL therefore adds several targeted optimizations that preserve the benefits of distributed shell execution while retaining recovery support.

5.1 Dynamic output persistence

Persistence is the optimization most directly associated with A3. It is helpful after faults because it lets the runtime reuse already produced data instead of recomputing entire upstream paths. It is also expensive because persisting every inter-subgraph edge adds extra local writes even when no fault occurs. FRACTAL addresses that tension with a per-subgraph policy instead of a single global switch.

The policy is guided by static information about the cluster and first-order properties of the workload. Subgraphs that operate on a single DFS block and have no downstream fan-out are treated as poor persistence candidates because their outputs are unlikely to survive the same worker failure that would force recovery [46]. Long-running upstream subgraphs with more expensive replay cost are better candidates. This dynamic choice is one of the central engineering tradeoffs in the system: the runtime buys future recovery speed only where the extra write path is likely to pay for itself.

5.2 Buffered I/O in the remote-pipe data path

The A2 remote pipe reader is performance-critical because every inter-subgraph byte crosses it. The reader therefore uses a bounded buffer with an 8-byte look-ahead for EOF-sentinel detection instead of scanning the whole stream or allocating fresh buffers for every chunk. Each iteration forwards all but the trailing 8 bytes, checks whether that trailing slice is the sentinel, and then reuses the small prefix as the starting state for the next read.

This design keeps the steady-state cost low while still supporting correct stream completion and offset accounting. It illustrates a broader point in FRACTAL: recovery support is only useful if the underlying runtime path remains efficient enough to preserve the gains of distributed execution.

5.3 Event-driven executor behavior

The executor runtime uses a lightweight event loop to manage staged subgraphs, completed tasks, and kill requests. The loop polls at a fixed interval, launches work up to a configurable concurrency limit, and avoids heavier synchronization where simple atomic state changes suffice. Completion events are deliberately tiny, and the worker-side control path is kept narrow. In the architecture view, this optimization strengthens the execution path downstream of A4: once the scheduler dispatches work, the executor has to keep that work moving with minimal control overhead.

Those choices matter because distributed shell workloads often contain many short-lived subgraphs. If the executor pays a large coordination cost per subgraph, the benefits of command-level parallelism erode quickly. The event-driven design is therefore as much about preserving the baseline speedups of DISH-style execution as it is about supporting replay.

5.4 Batched scheduling

Another important cost center is the A4 scheduler. Scripts over large or highly partitioned inputs can generate many subgraphs, and distributing them one by one would amplify both network chatter and control-plane overhead. Fractal batches subgraphs with the same target and sends them asynchronously, reducing per-subgraph scheduler cost and improving scalability as the cluster widens.

The tradeoff is familiar: more aggressive batching reduces control-plane overhead, but it also makes the runtime's unit of dispatch less granular. Fractal uses batching where it improves scheduling efficiency without changing the underlying recovery unit, which remains the subgraph.

6 Evaluation

We evaluate FRACTAL along three dimensions that are central to its runtime design: fault-free overhead on the common path, recovery behavior under worker faults, and the tradeoff enabled by dynamic persistence.

6.1 Methodology

Baselines. We compare FRACTAL against three baselines. Bash represents sequential execution on a single node [52]. Apache Hadoop Streaming represents a MapReduce-backed system that offers fault-tolerant execution of arbitrary binaries, including black-box Unix commands, but expects mapper/reducer wrappers [22]. DISH represents a state-of-the-art system for automatic shell-script distribution that lacks fault tolerance and therefore serves as the closest point of comparison on the fault-free path [39].

This set of baselines separates three questions. Compared with Bash, it measures whether distribution is worthwhile at all. Compared with DISH, it captures the incremental cost of fault-tolerance support on the fast path. Compared with Hadoop

Streaming, it shows whether Fractal's replay model improves recovery time without requiring users to port shell scripts into a different programming abstraction.

The benchmark sets in Tab. 2 follow the same workload families used in the Fractal evaluation materials [28, 31, 39, 56]. Together they span 77 scripts and 547 lines of shell code. That diversity matters because a runtime that looks good only on long, regular dataflow jobs would miss a large fraction of actual shell workloads. Classics [1, 2, 27, 35, 51] and Unix50 [3, 40] cover command-heavy Unix pipelines, NLP [6] contributes text-processing tutorial workloads, Analytics [55, 58] contributes data-processing scripts, and Automation [43, 45, 47] contributes administration, media-processing, and network-analysis tasks. As in the original Fractal evaluation, AHS comparisons are restricted to the suites expressible with mapper/reducer wrappers.

Tab. 2: Benchmark summary. Summary of all the benchmarks used to evaluate FRACTAL and their characteristics.

Benchmark	Scripts	LoC	AHS	Input
Classics	10	103	✓	3 GB
Unix50	34	34	✓	10 GB
NLP	22	280	✘	10 GB
Analytics	5	62	✓	33.4 GB
Automation	6	68	✘	2.1-30 GB

Hardware, deployment, and fault scenarios. FRACTAL is evaluated on two clusters. One is a 30-node CloudLab deployment with 8-core Intel Xeon D-1548 machines, 64 GB of RAM, local NVMe, and 10 Gb networking. The other is a 4-node Raspberry Pi 5 cluster with 4-core Arm CPUs, 8 GB of RAM, SSD storage, and 1 Gb networking. Both use Ubuntu 22.04-based Docker images and HDFS-backed storage.

Fault scenarios include both manually induced worker failures and softer, automated failures triggered by the evaluation harness. The `frac` tool is used as part of that experimental setup to inject failures at controlled execution points, making it possible to characterize the runtime's recovery behavior systematically.

The key recovery configurations used later are `regular` faults, `merger` faults, and microbenchmarks that vary output persistence. That split matches the runtime design. Regular and merger failures exercise different parts of the replay logic, while the persistence microbenchmark isolates one of the runtime's most important implementation tradeoffs.

6.2 Fault-free execution

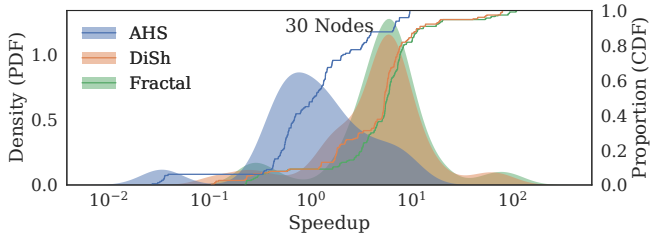

Fig. 3: Fault-free performance summary. On the 30-node cluster, FRACTAL remains close to DISH while substantially outperforming Hadoop Streaming on the benchmark subset expressible with mapper/reducer wrappers.

Fractal's fault-tolerance support does not erase the benefit of distributed shell execution. As shown in Fig. 3 and Tab. 3, FRACTAL remains close to DISH on the fault-free path and materially ahead of Hadoop Streaming. On the 30-node cluster, the average speedup over Bash is 9.64× for FRACTAL versus 8.20× for DISH and 1.99× for AHS on the supported subset of scripts. These results show that the runtime mechanisms described in Sections 4 and 5 remain compatible with the scale-out benefits that motivate distributed shell execution in the first place.

These results follow from the same implementation choices described earlier. The persistence path avoids a more expensive "tee everything to disk" design, the event loop and batching keep control-plane cost low, and the overall instrumentation footprint is small enough that short critical paths are not overwhelmed by recovery bookkeeping. The remaining slowdowns occur where the workloads are already poor scaling candidates, such as scripts dominated by near-instant commands or wide fan-in aggregation.

The summary in Tab. 3 makes the same point numerically across both cluster sizes. Fractal remains close to DiSH on the common path while maintaining a large advantage over Hadoop Streaming on the supported subset. The gap is widest on larger clusters, where the distributed execution benefits are strongest and the incremental bookkeeping cost of fault tolerance remains comparatively small.

Tab. 3: Fault-free performance comparison highlights. Average, minimum, and maximum speedups over Bash for FRACTAL, DiSH, and AHS across all benchmarks.

	4 Node			30 Node		
System	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
FRACTAL	5.93	0.28	18.55	9.64	0.22	107.8
DiSH	5.88	0.15	19.04	8.20	0.10	78.35
AHS	1.27	0.01	6.94	1.99	0.02	9.48

6.3 Recovery behavior

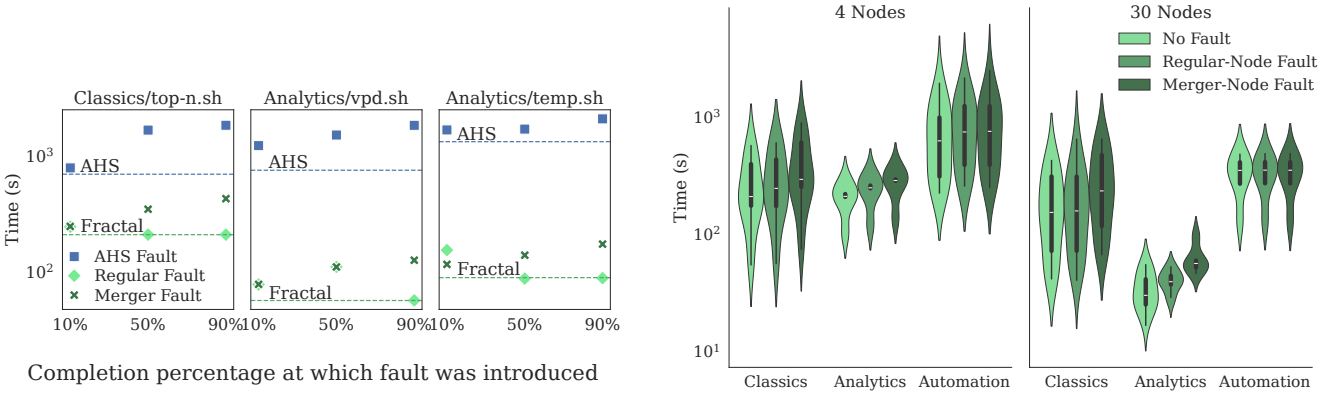

Fig. 4: Recovery behavior. Left: compared with Hadoop Streaming, Fractal's recovery times remain much closer to the fault-free runtime because the runtime replays only the affected subgraphs instead of restarting larger task stages. Right: soft-fault experiments on 4-node and 30-node clusters show that merger faults are often costlier than regular-worker faults, but both remain close to the fault-free execution time compared with coarse-grained rerun strategies.

The recovery results are the clearest evidence that the runtime design is doing the intended work. Fig. 4 compares Fractal with Hadoop Streaming and then breaks recovery cost down by benchmark family and worker role. Fractal's recovery remains much closer to the no-fault runtime because the coordinator replays only the affected subgraphs and reconnects downstream communication at the recorded byte offsets. In contrast, the stage-oriented recovery model in Hadoop Streaming loses more completed work.

FRACTAL recovers within roughly 1.26× of the fault-free runtime and yields a 7.8–16.4× speedup over Hadoop Streaming in the representative hard-fault experiments. This result follows directly from the runtime machinery described earlier. Progress tracking identifies what finished. remote pipe offsets preserve downstream correctness. Persistence lets the runtime skip some expensive upstream recomputation. Without those three pieces working together, the recovery advantage would collapse into a coarse rerun.

Fig. 4 also highlights a second point that is especially relevant to the runtime architecture: different worker roles fail differently. Regular-worker faults often interfere with fewer shared dependencies and therefore recover faster. Merger faults are typically more expensive because they sit near aggregation boundaries and may require more upstream coordination. That asymmetry is not incidental; it reflects the fact that Fractal's recovery unit is the subgraph, not an undifferentiated worker process.

6.4 Dynamic persistence microbenchmark

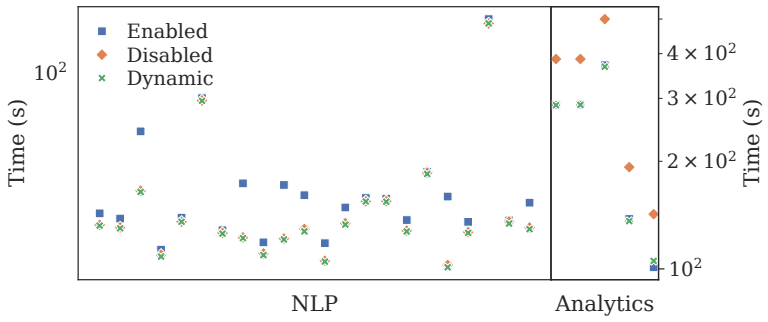

Fig. 5: Dynamic persistence microbenchmark. Persistence hurts short, fault-free NLP workloads but helps faulted Analytics workloads; the runtime heuristic chooses the right side of that tradeoff in both cases.

Fig. 5 isolates one of the most important implementation choices in the runtime. Always persisting outputs is too expensive for short workloads that would not benefit from reusable cached data. Never persisting outputs is too expensive when a fault would otherwise force long upstream recomputation. Enabling persistence adds about 21.0% overhead on the fault-free NLP benchmark, while disabling persistence adds about 38.7% overhead on the faulted Analytics benchmark.

This is precisely the scenario that justifies Fractal's per-subgraph persistence policy. The policy is simple by design, but the microbenchmark shows that even first-order heuristics can recover a substantial fraction of the ideal choice. It documents a practical engineering decision, the workload regimes where it matters, and the measurable consequences of getting it wrong.

6.5 Runtime-centered takeaways

Taken together, the evaluation supports three high-level conclusions. The runtime's recovery support preserves most of the fault-free advantage of distributed shell execution, selective replay at subgraph granularity is materially better than coarse rerun strategies for worker faults, and dynamic persistence is not a cosmetic optimization but one of the mechanisms that lets Fractal reconcile fault tolerance with competitive steady-state performance.

7 Related Work

7.1 Distributed shells and shell tooling

Classic Unix job-distribution tools such as Sun Grid Engine and GNU Parallel help users spread work across machines, but they rely on manual orchestration and do not provide Fractal's style of transparent recovery [19, 49]. Systems such as Rc, Dgsh, and gsh extend shell composition in useful ways, but they either require rewriting or do not target distributed recovery [12, 34, 47]. Recent automated systems including PaSh, POSH, and DiSh are the most immediate precursors to Fractal because they already operate on unmodified shell scripts and expose command-level parallelism and distribution [28, 39, 43, 56]. Fractal differs by adding runtime support for worker-fault recovery while retaining that unmodified-shell interface.

7.2 Distributed data-processing frameworks

MapReduce, Hadoop, Spark, Dryad, Ray, Naiad, CIEL, and related systems all provide important reference points for distributed recovery and replay [10, 24, 36–38, 42, 48, 57, 59]. However, they expect users to express computation in framework-native tasks, operators, or dataflow APIs. Hadoop Streaming and Dryad Nebula get closer because they can invoke black-box binaries, but they still do not preserve general shell semantics or automatic handling of dynamic shell behavior [22, 24].

The key difference is therefore not only that Fractal tolerates faults, but *where* it does so. Its recovery logic lives inside a runtime specialized for shell-generated subgraphs, ad-hoc byte streams, and selective replay boundaries rather than inside a generic data-processing abstraction.

7.3 Other fault-tolerance substrates

Checkpointing and replication techniques at the VM, container, and serverless layers provide another point of comparison [8, 9, 13, 26, 29, 32, 33, 60]. Those systems can mask failures, but they generally depend on infrastructure support, logging, or framework-managed state and do not reason about shell-level dependencies. The gg system is notable for scaling black-box commands to serverless functions, yet it relies on retry rather than Fractal's pipeline-aware replay model [16].

8 Limitations

Fractal's runtime is intentionally scoped to worker faults because those are the failures that most directly threaten long-running distributed shell executions. The coordinator is assumed to remain available, and replication of coordinator state is left to future work that could be handled with standard consensus techniques such as Raft [41]. That assumption keeps the prototype and the evaluation focused, but it also means that Fractal is not yet a complete answer to all control-plane failures.

The replay boundary is also conservative. Side-effectful commands that mutate external state remain outside the distributed replay path and execute in the client-side `main` subgraph. This is the right correctness choice for the current prototype, but it limits how much of a script can benefit from the runtime when a workflow contains rich external effects. One possible extension would combine Fractal's subgraph replay with a sandbox or rollback substrate such as TRY so that a larger class of side-effectful fragments could be recovered safely [54].

Finally, the current single-coordinator design scales comfortably to the evaluated deployments, but it may not remain the right control-plane structure for substantially larger or denser clusters. If Fractal were pushed beyond the current regime, scheduler sharding or a hierarchical control plane would likely become part of the runtime story as well.

9 Conclusion

We presented the runtime design and implementation of FRACTAL, focusing on the mechanisms that make worker-fault recovery practical for distributed shell scripts: subgraph-level replay boundaries, progress and discovery services, exactly-once remote pipe delivery, distributed input access, executor-side control flow, and the optimizations that keep these mechanisms inexpensive on the common path. The result is a runtime story centered on selective replay rather than on coarse rerun.

These results show that fault tolerance for shell scripts is not just a matter of adding retries around distributed execution. The shell's black-box commands, ad-hoc byte streams, dynamic behavior, and side effects force the runtime to keep much closer track of what has actually happened at subgraph boundaries. Fractal's evaluation suggests that this extra bookkeeping is worthwhile: the system remains close to state-of-the-art fault-free shell distribution while recovering substantially faster than a coarser stage-based baseline.

More broadly, these results show that shell-specific fault tolerance depends on careful runtime engineering as much as on high-level recovery policy. Progress tracking, replay-safe communication, and selective persistence together make it possible to retain the advantages of distributed shell execution while substantially reducing the cost of worker faults.

Acknowledgements

I thank Nikos Vasilakis for advising me on this project and report. I am grateful to Zhicheng Huang for his collaboration on the design, implementation, and evaluation of Fractal; this report focuses on my runtime-centered contributions, but many parts of the broader system were developed jointly. I thank Yizheng Xie and Konstantinos Kallas for their contributions to the Fractal project and for feedback on the paper and prototype. I also thank Nikos Pagonas, the Brown Atlas group, the Brown Systems Group, the Brown CS2952R (Fall'24) participants, and our collaborators at Stevens Institute of Technology for discussions and feedback. This material is based upon research supported by NSF awards CNS-2247687, CNS-2312346, and CCF-2340479; DARPA contract no. HR001124C0486; a Fall'24 Amazon Research Award; a Google ML-and-Systems Junior Faculty award; a seed grant from Brown University's Data Science Institute; and a BrownCS Faculty Innovation Award.

References

[1] Jon Bentley. Programming pearls: a spelling checker. *Commun. ACM*, 28(5):456–462, may 1985.

[2] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: a literate program. *Commun. ACM*, 29(6):471–483, jun 1986.

[3] Pawan Bhandari. Solutions to unixgame.io, 2020. Accessed: 2020-04-14.

[4] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, aug 2017.

[5] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Kenneth Ward Church. Unix for poets, 1994. Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods.

[7] CRIU community. Checkpoint/restart in userspace (criu). https://criu.org/, 2019. Accessed: April 2025.

[8] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 105–120, 2015.

[9] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX symposium on networked systems design and implementation*, pages 161–174. San Francisco, 2008.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, jan 2010.

[12] Tom Duff. Rc: A shell for plan 9 and unix systems. *AUUGN*, 12(1):75, 1990.

[13] George W Dunlap, Dominic G Lucchetti, Michael A Fetterman, and Peter M Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2008.

[14] Johan Eveleens and Chris Verhoef. The rise and fall of the chaos report figures. *IEEE software*, 27(1):30–36, 2009.

[15] Bent Flyvbjerg and Alexander Budzier. Why your it project might be riskier than you think. *arXiv preprint arXiv:1304.0265*, 2013.

[16] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 475–488, 2019.

[17] Aeleen Frisch. *Essential system administration: Tools and techniques for linux and unix administration*. " O'Reilly Media, Inc.", 2002.

[18] Ishaan Gandhi and Anshula Gandhi. Lightening the cognitive load of shell programming. *PLATEAU 2020*, 2020.

[19] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36. IEEE, 2001.

[20] GitHub. The state of the octoverse 2024: The most popular programming languages, 2024. Accessed: 2024-10-31.

[21] Michael Greenberg. Word expansion supports posix shell interactivity. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 153–160, 2018.

[22] Hadoop. Hadoop streaming. https://hadoop.apache.org/docs/r3.4.0/hadoop-streaming/HadoopStreaming.html, 2024. [Online; accessed June 13, 2024].

[23] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 38–49, 2020.

[24] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on computer systems 2007*, pages 59–72, 2007.

[25] Jeroen Janssens. *Data science at the command line: Facing the future with time-tested tools*. " O'Reilly Media, Inc.", 2014.

[26] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

[27] Dan Jurafsky. Unix for poets, 2017. Accessed: 2024-09-16.

[28] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, just-in-time shell script parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–18. USENIX Association, July 2022.

[29] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve:{Execute-Verify} replication for {Multi-Core} servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, 2012.

[30] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[31] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The koala benchmarks for the shell: Characterization and implications. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25)*, pages 449–64, Boston, MA, July 2025. USENIX Association.

[32] David H Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. Doing more with less: Orchestrating serverless applications without an orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1505–1519, 2023.

[33] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110, 2009.

[34] Chris McDonald and Trevor I Dix. Support for graphs of processes in a command interpreter. *Software: Practice and Experience*, 18(10):1011–1016, 1988.

[35] Malcolm D. McIlroy, Elliot N. Pinson, and Berkley A. Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[36] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[37] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[38] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. {CIEL}: A universal execution engine for distributed {Data-Flow} computing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[39] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. DiSh: Dynamic Shell-Script distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 341–356, Boston, MA, April 2023. USENIX Association.

[40] Nokia Bell Labs. The unix game—solve puzzles using unix pipes, 2019. Accessed: 2020-03-05.

[41] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[42] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[43] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A data-aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631, 2020.

[44] Arnold Robbins and Nelson HF Beebe. *Classic Shell Scripting: Hidden Commands that Unlock the Power of Unix*. " O'Reilly Media, Inc.", 2005.

[45] Michael Schröder and Jürgen Cito. An empirical investigation of command-line customization. *Empirical Software Engineering*, 27(2), December 2021.

[46] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[47] Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[48] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. Jetstream: a self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pages 1–8, 2015.

[49] Ole Tange. Gnu parallel-the command-line power tool. *Usenix Mag*, 36(1):42, 2011.

[50] Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. No Starch Press, 2004.

[51] Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. No Starch Press, 2004.

[52] The Free Software Foundation. Bash shell, 2009. [Online; accessed 30-October-2024].

[53] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, 2014.

[54] Try Authors. try: Inspect a command's effects before modifying your live system. https://github.com/binpash/try, 2023. Version 0.2.0, accessed August 13, 2025.

[55] Eleftheria Tsaliki and Diomidis Spinellis. The real statistics of buses in athens, 2021.

[56] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[57] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[58] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[60] 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)*, pages 1187–1204, 2020.