ARTIFACT EVALUATED USENIX AVAILABLE
ARTIFACT EVALUATED USENIX FUNCTIONAL
ARTIFACT EVALUATED USENIX REPRODUCED

# Fractal: Fault-Tolerant Shell-Script Distribution

Zhicheng Huang*
*Brown University*

Ramiz Dundar*
*Brown University*

Yizheng Xie
*Brown University*

Konstantinos Kallas
*University of California, Los Angeles*

Nikos Vasilakis
*Brown University*

## Abstract

This paper presents FRACTAL, a new system that offers fault tolerant distributed shell script execution for unmodified scripts. FRACTAL first distinguishes recoverable regions from side-effectful ones, and augments them with additional runtime support aimed at fault recovery. It employs precise dependency and progress tracking at the subgraph level to offer sound and efficient fault recovery. It minimizes the number of upstream regions that are re-executed during recovery and ensures exactly-once semantics upon recovery for downstream regions. Evaluation on 4- and 30-node clusters indicates average fault-free speedups of (1) >9.6× over Bash, a single-node shell-interpreter baseline, (2) >5.5× over Hadoop Streaming, a MapReduce system that supports language-agnostic third-party components, and (3) 17% over DISH, a state-of-the-art fault-intolerant shell-script distribution system—all while recovering 7.8–16.4× faster than Hadoop Streaming in cases of faults.

## 1 Introduction

The Unix shell remains the 8th most popular language on GitHub in 2024 [21], widely used for a variety of workloads [18, 29, 31, 56, 63]. Its popularity can be attributed to several characteristics, including (1) language-agnosticism, flexibly composing an arsenal of task-specific components available in a variety of languages, and (2) dynamism, providing features such as command substitution, variable expansion, and file system reflection.

Unfortunately, the shell's characteristics complicate *fault-tolerant* shell-script scale-out. The black-box nature of third-party components hinders internal state tracking, complicating recovery after node faults and limiting scale-out opportunities. Dynamic behaviors and arbitrary side effects make re-executing failed subgraphs challenging, affecting the correctness of re-executed scripts. Tolerating faults is often at odds with maintaining the shell's expressiveness, without forcing users to modify existing (often legacy) scripts.

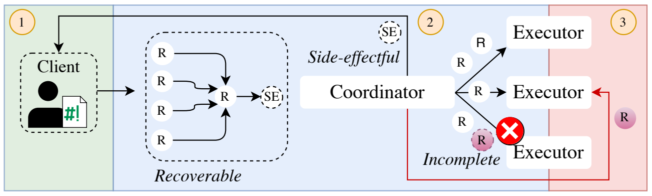


**Fig. 1: FRACTAL's high-level workflow.** FRACTAL (1) isolates side-effectful regions from recoverable regions; (2) executes recoverable subgraphs on nodes, tracking locality, dependencies, progress, and health; (3) detects faults, re-scheduling the minimal set of unfinished subgraphs for re-execution.

Typical approaches such as checkpointing [5, 6, 8, 37, 66], barriers [11, 12, 26], and lineage [30, 74] are ill-suited for the setting of the shell (Table 1). As a result, while research on and around the shell is exploding [24, 25, 34, 42, 51, 54, 70], currently no system tolerates faults—*e.g.*, worker aborts or network partitions—during distributed shell-script execution.

**Fault-tolerant distribution with Fractal**: FRACTAL is a system supporting fault-tolerant shell-script distribution: it operates on existing shell scripts without modification, supports all shell's dynamic features, allows language-agnostic composition of black-box components, and enables side-effect-free distribution with recovery from worker faults.

FRACTAL (Fig. 3) first uses PaSh-JIT [34], a just-in-time compiler, to transform a POSIX shell script into a data-flow graph (DFG). FRACTAL's coordinator then uses command annotations to identify side-effectful commands unsafe for fault-tolerant distribution, to be executed on the client's node. Next, for all regions safe for fault-tolerant distribution, it wraps inter-subgraph edges with a lightweight remote-pipe primitive that records byte-level progress and enforces exactly-once semantics. FRACTAL's per-subgraph heuristic decides whether to persist outputs locally to balance runtime overhead and recovery speed. At runtime, after subgraphs are scheduled to

[1]Authors contributed equally. Zhicheng is now with the University of California, San Diego. Ramiz is now with Google.

**Tab. 1:** Comparison of fault tolerance mechanisms across key desiderata in shell scripts.

| Desideratum | Checkpointing [5,6,8,37,66] | Barrier-based [11,12,26] | Lineage-based [30,74] | FRACTAL |
|---|---|---|---|---|
| D1 Handles Black-Box State | No | Yes | No | Yes |
| D2 Ad-Hoc Pipe Streaming Integrity | No | No | No | Yes |
| D3 Side-Effect Management | Partial | No | No | Yes |
| D4 Dynamism Compatibility | Partial | No | No | Yes |
| D5 Recovery Granularity | Coarse | Coarse | Fine | Fine |
| D6 No Script Modification | Partial | Partial | No | Yes |

executor nodes, FRACTAL's health and progress monitors continuously track inter-subgraph dependencies and byte-level delivery to detect faults. When a fault occurs, the coordinator computes the minimal set of subgraphs that need to be replayed. By re-executing only these subgraphs, FRACTAL eliminates redundant work and guarantees exactly-once semantics across all downstream regions.

FRACTAL also introduces a new subsystem, **`frac`** (for *fracture*), that injects runtime faults in large scale distributed deployments, enabling fault simulations in an evaluation environment. **`frac`** can be used independently of FRACTAL and is therefore released as a separate tool.

**Key results**: In fault-free scenarios, FRACTAL achieves substantial performance improvements, delivering an average speedup of 9.6× over Bash, a standard shell interpreter, 5.5× over Apache Hadoop Streaming (AHS), a cluster-computing system incorporating black-box Unix commands, and 17% over DISH, a recent *fault-intolerant* scale-out system.

FRACTAL recovers from faults within 1.26× of the script's fault-free runtime, achieving a 9.3× speedup over AHS—while supporting improved expressiveness and requiring no manual modifications to the source programs.

**Paper outline and contributions**: The paper starts with a discussion about the design landscape for fault-tolerant shell-script distribution (§2). It then introduces FRACTAL's design through a motivating example (§3). It proceeds with FRACTAL's key subsystems (§4–§6):

- *Execution engine* (§4): FRACTAL's remote-pipe instrumentation, progress and health monitors, and executor runtime enable efficient and precise recovery.
- *Performance optimizations* (§5): FRACTAL's targeted optimizations, including event-driven execution, buffered I/O, and batched scheduling, minimize critical-path overhead.
- *Fault injection* (§6): FRACTAL's **`frac`** tool enables precise, large-scale fault injections to characterize recovery behavior under real-world conditions.

The paper then presents FRACTAL's evaluation (§7), related work (§8), limitations (§9), and conclusion (§10).

**FRACTAL's fault model**: FRACTAL assumes that *worker nodes* may crash in either fail-stop or fail-restart fashion, mirroring typical large-scale deployments on commodity hardware. These crashes afflict the long-running, data-parallel subgraphs where recomputation is expensive, so tolerating them yields the greatest practical benefit. Resilience of the coordinator, client shell, and storage master (*e.g.*, the HDFS NameNode) is outside our scope and can be provided by well-established primary-backup or consensus replication [27]. Side-effectful commands are confined to a special small subgraph on the client node and usually perform quick pre- or post-processing tasks such as `mv`, logging, or summary writes. If the client node fails, rerunning the script repeats only these minor operations, keeping the overall recovery cost negligible. By focusing on frequent worker faults and delegating rarer fault modes to orthogonal techniques, FRACTAL remains lightweight yet broadly useful.

**Availability**: FRACTAL's implementation and evaluation is publicly accessible as an MIT-licensed open-source project at https://github.com/binpash/fractal.

## 2 Fault Tolerance for Shell Script

This section begins by outlining the desiderata for fault-tolerant shell-script distribution (Table 1, col. 1), derived from the shell's unique characteristics. It then examines the limitations of existing fault-tolerance mechanisms in meeting these desiderata (Table 1, cols. 2-4). Finally, it presents FRACTAL's design and how it meets these desiderata (Table 1, col. 5).

### 2.1 Desiderata

Shell scripts uniquely blend diverse commands, streaming pipelines, flexible control flow, and dynamic expansion at runtime. While providing unmatched expressiveness and simplicity, it complicates fault-tolerant execution. To guide our design, we identify six fault-tolerance desiderata that any robust shell-script distribution mechanism must satisfy.

D1 **Black-box state handling**: Shell pipelines invoke external binaries (*e.g.*, `sort`, `grep`, `unzip`) whose internal state cannot be inspected or checkpointed. Such commands may hold gigabytes of data internally without any API for partial snapshots, making it impossible to selectively roll back and resume. A fault-tolerance scheme must recover progress without requiring analysis of or hooks in these opaque commands.

D2 **Ad-hoc pipe streaming integrity**: Shell commands communicate via unstructured byte streams over ad-hoc

UNIX pipes, with arbitrary buffering, chunking, and per-command transformation semantics. faults leave no record of how many bytes or which logical "records" were consumed, and replaying an opaque stream risks duplicating or dropping data. Under faults, the system must guarantee exactly-once delivery so that no data is lost or duplicated despite retries.

D3 **Side-effect management**: Shell commands often perform non-idempotent external actions (*e.g.*, appending to files, making network calls, updating system states) that modify the environment. Simply re-running a partially completed side-effectful command can append data again or re-trigger external actions. Recovery must prevent repeated side effects or orphaned partial writes.

D4 **Dynamism compatibility**: Shell scripts are highly dynamic, by which their behavior (*e.g.*, control flow, executed commands, *etc.*) critically depends on the values of environment variables and the state of the file system at runtime. A fault-tolerance design must handle these dynamic behaviors and not assume that a script's computation can be determined statically and ahead-of-time.

D5 **Fine recovery granularity**: Shell pipelines often chain many long-running commands, so re-executing whole stages or other coarse-grained recovery units wastes substantial work. Achieving fine-grained replay is further complicated by black-box commands with opaque internal state and by ephemeral outputs already consumed downstream. Therefore, a robust recovery mechanism must isolate and replay only the minimal set of affected subgraphs of the workflow.

D6 **No script modification**: Shell scripts are notoriously difficult to program and maintain [19, 22], and modifying or re-implementing legacy scripts can be costly and error-prone [15, 16]. Thus, forcing rewrites is prohibitive. An ideal solution should preserve existing scripts unchanged, transparently adding recovery support.

## 2.2 Existing Approaches

We examine three fault-tolerance paradigms—checkpointing, barrier-based, and lineage-based—and how each falls short of the shell-script desiderata. It is worth noting that, *in practice, failing to meet even one of the desiderata may be enough to disqualify a system as a viable solution for shell scripts.*

**Checkpointing**: Checkpointing systems snapshot process or operator state [5, 6, 8, 37, 66], usually by exposing runtime hooks in each operator (*e.g.*, in streaming engines or controlled process trees). in streaming engines or controlled process trees. D1 Checkpointing needs APIs such as `get-processing-state` [6] and `getState` [48] to retrieve internal state, impossible for opaque, language-agnostic shell commands. Incremental or per-operator snapshots reduce overhead but still require instrumentation inside each command, which is impossible for opaque shell binaries. D2 Frameworks such as Flink [5], Storm [66], and Kafka [37] embed barriers or offset-tracked logs, but UNIX pipes lack both, so checkpointing byte streams would require external brokers. D3 Transactional sink APIs (*e.g.*, Flink's `TwoPhaseCommitSinkFunction`) manage side-effectful writes within the framework, but shell scripts perform arbitrary file and network I/O outside of any transaction boundary. D4 Checkpointing assumes a static operator set, hampering recovery when new processes appear mid-execution. D5 Full snapshots capture entire pipelines or process trees, imposing high overhead and causing unnecessary re-execution for chains of short-lived commands; per-command checkpointing would introduce prohibitive runtime overhead and complex coordination. D6 Adopting checkpointing for shell scripts demands wrapping or replacing every invocation, violating the no-modification requirement for legacy or ad-hoc scripts.

**Barrier-based**: Barrier-based systems (*e.g.*, MapReduce [11, 12]) achieve fault tolerance by retrying entire map or reduce tasks upon faults, relying on a static task graph. D1 While this model supports black-box tasks, it lacks the ability to resume partially completed computation: failed components must restart from scratch, even if most work had completed. D2 Barrier-based models are not ideal for streaming data; their retry model simply replays upstream outputs, leading to potential duplication and breaking exactly-once guarantees. Streaming extensions (*e.g.*, Kafka Streams [37]) guarantee exactly-once by buffering entire micro-batches or writing to durable topics—adapting raw UNIX pipes would mean replacing each `|` with a brokered topic, leading to serialization and network hops for every pipeline edge and head-of-line blocking at batch boundaries, undermining the shell's low-latency, in-memory streaming model. D3 Barrier-based retries rerun every command in a failed task including any non-idempotent side-effectful commands such as file appends or HTTP calls. Because there is no transactional or deduplication API at the shell level, each retry re-issues side-effects. Avoiding duplicates requires invasive wrappers or custom idempotency logic for every shell command, violating transparent execution. D4 The static task graph must be fully specified before execution; shell scripts that spawn new commands via loops or `eval` cannot be dynamically incorporated, leaving on-the-fly pipelines untracked. D5 Recovery granularity is fixed at task boundaries; defining each shell command as its own task could narrow scope but forces scripts to be restructured into dozens or hundreds of map/reduce jobs, incurring prohibitive scheduling overhead. D6 Finally, while MapReduce does not mandate full rewrites, it still requires structuring logic into map and reduce phases—limiting flexibility and imposing extra effort when adapting existing or evolving shell scripts. Hadoop Streaming [26] supports arbitrary binaries but still forces explicit mapper/reducer wrappers, violating the no-modification desideratum.

**Lineage-based**: Lineage-based systems (*e.g.*, Dryad [30] and Spark [74]) record operator DAGs and only replay failed tasks. While effective for deterministic dataflows in one framework, they struggle with ad-hoc shell pipelines. D1 Lineage frameworks assume each operator is a pure function of its visible inputs and outputs, but black-box shell commands (e.g., `sort`, `uniq`) buffer and transform data internally without producing retrievable artifacts, so lineage cannot reconstruct progress. D2 Spark Streaming's micro-batch and offset model cannot fit unbounded UNIX pipes, so it would require rewriting each pipe as a Spark stage. D3 Lineage frameworks mitigate side-effects via transactional sinks only if every write goes through their API, but shell commands perform arbitrary I/O (*e.g.*, `>>`, `mv`) outside of any transaction boundary, requiring invasive wrappers to prevent duplicates. D4 Dynamic DAG registration requires user callbacks (*e.g.*, `writeStream` in Spark Structured Streaming), whereas shell loops and `eval` spawn processes silently, leaving lineage unnotified and unprepared to recover new branches. D5 They support granular recomputation: if tasks and dependencies are captured, they recompute only failed partitions and dependencies. D6 Lineage approaches require rewriting scripts into deterministic, functional forms that match the lineage mode, cumbersome for legacy or rapidly evolving scripts.

## 2.3 Our approach

At its core, FRACTAL treats *a program subgraph* as the atomic unit of computation, because shell commands are isolated processes whose pipes and files are declared explicitly, so their dependencies are visible without deep analysis. This design avoids costly instrumentations of every individual shell command while remaining fine-grained enough to avoid large-scale re-execution. At subgraph boundaries, FRACTAL injects minimal runtime primitives that transparently track progress and enable precise recovery without affecting the internal logic of any black-box command.

This model addresses the key challenges of shell-script fault tolerance. D1 By tracking only inputs and outputs for each subgraph, we never peek inside a command's memory or file descriptors. Each command remains unmodified; recovery works solely from its byte-stream boundaries. D2 Byte-level progress tracking guarantees no data loss or duplication, even when a subgraph mixes streaming filters with blocking operators. D3 Commands with non-idempotent side effects remain in a special subgraph under user control; distributed subgraphs perform only pure data transformations or write to isolated files that can be atomically swapped in upon success. D4 Subgraphs are derived at compile time from the AST, so any new commands—spawned via loops, conditionals, or environment expansions—automatically become first-class fault-tolerance units without requiring a static representation. D5 FRACTAL recovers at the subgraph level, not per-command. Because the data-flow compiler inflates each pipeline into orders-of-magnitude more black-box commands, granular recovery at the command-level would force every one of them to be wrapped, coordinated, and potentially rolled back synchronously, incurring prohibitive overhead. Subgraph-level recovery strikes a sweet spot: small enough to avoid re-doing large amounts of work, yet coarse enough to amortize the runtime instrumentation cost. D6 All fault-tolerance logic is injected by the compiler. Users run unmodified POSIX shell scripts under FRACTAL, without needing to reexperss their script in constrained APIs.

While the focus is correct and efficient recovery, FRACTAL also aims to deliver *near state-of-the-art* performance in fault-free executions.

```
1  #!/bin/bash
2  in=${in:-$TOP/log-analysis/nginx-logs}
3  out=${out:/outputs}
4  bots='Googlebot|Bingbot|Baiduspider|Yandex|'
5  mkdir $out && hdfs dfs -mkdir /log-analysis
6
7  # 1. Download and store nginx logs to HDFS
8  wget "$SOURCE/data/nginx-logs.zip"
9  unzip nginx-logs.zip && rm nginx-logs.zip
10 hdfs dfs -put nginx-logs /log-analysis/nginx-logs
11
12 # 2. Analyze log files
13 for log in $(hdfs dfs -ls -C $in); do
14   name="$out/$(basename "$log".log)"
15   # 3. Identify bot IPs by visit frequency
16   hdfs dfs -cat "$log" | grep -E $bots | cut -d" " -f1 |
17        sort | uniq -c | sort -rn  >> "${name}.out"
18   # Further analysis omitted for brevity...
19 done
```

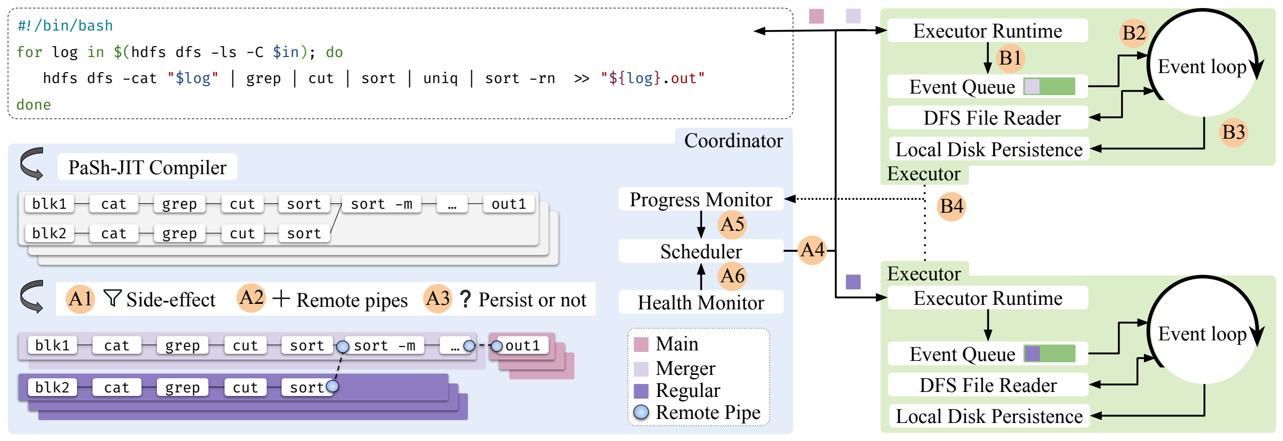
**Fig. 2: Log analysis script (*Cf.*§3).** The script downloads Nginx logs, stores them on a distributed filesystem, and analyzes them to extract traffic statistics—slightly modified from POSH [54] to highlight idiomatic shell challenges.

## 3 Example and Overview

Scripts that process large datasets usually need to interact with a distributed file system (DFS) such as HDFS [58], NFS [28], or Alluxio [41], as their input data does not fit on a single computer. FRACTAL scales out the computation to facilitate data locality, data parallelism, and pipeline parallelism, while ensuring recoverability when a participating node fails.

**Example script and problem**: Fig. 2 presents an example shell script analyzing Nginx logs, divided into three parts: (1) setup ($L_{7-10}$), downloading 150GB of logs and storing them on HDFS; (2) driver ($L_{13-14}$, $L_{19}$), iterating over the HDFS directory, piping log files to the analysis pipeline and appending results to a dynamically determined local file; and (3) analysis ($L_{15-18}$), identifying known bot IPs by visit frequency.

A developer opting for distributed execution, either manual or more recently automated [51, 54], is left with only one option when a compute node fails: to restart the entire computa-



```
#!/bin/bash
for log in $(hdfs dfs -ls -C $in); do
   hdfs dfs -cat "$log" | grep | cut | sort | uniq | sort -rn  >> "${log}.out"
done
```

**Fig. 3: FRACTAL's architecture.** From a client shell script, FRACTAL uses PaSh-JIT to build a DFG, applies annotations to isolate the unsafe `main` subgraph (A1), and splits the rest into `regular` and `merger` subgraphs at HDFS block boundaries. It then instruments each edge with remote-pipe primitives (A2) and uses a lightweight heuristic to persist outputs per subgraph (A3). The coordinator schedules subgraphs (A4) and leverages the progress (A5) and health (A6) monitors to re-execute only failed subgraphs. Executors reconstruct subgraphs into shell scripts and run them in a tight, non-blocking event loop (B1-4), streaming data via remote pipes, the distributed file reader, and local cache.

tion. Unfortunately, such a restart impacts both *performance*, as a full rerun will waste over 3 hours, and *correctness*, as the script appends to a file and thus upon fault may result in partial outputs—worse even, potentially mixed with correct results from earlier fault-free executions.

**Challenges for fault tolerance**: The script in Fig. 2 illustrates why fault tolerance is so challenging: it invokes black-box commands (*e.g.*, `uniq -c` at $L_{17}$) whose internal counters cannot be inspected or checkpointed ( D1 ), passes data through chains of ephemeral UNIX pipes (*e.g.*, from `grep` to `sort`, then to `uniq`) with no built-in barriers or offsets ( D2 ), and relies on side-effectful operations (*e.g.*, the append operator >> at $L_{17}$) that risk duplication or partial writes upon retry ( D3 ). Control-flow constructs such as `for log in $IN/*.log` spawn commands dynamically based on variable values, preventing any static view of the computation graph ( D4 ). Re-executing the entire script on 150GB of logs takes over 3 hours, so a coarse-grained rerun is prohibitively expensive ( D5 ). Finally, such scripts are often legacy or incrementally maintained, so frameworks requiring modifying them are error-prone and impractical. ( D6 ).

**Fractal overview**: Fig. 3 presents an overview of FRACTAL. FRACTAL builds on PaSh-JIT [34] to retrieve a DFG representation of the target script. FRACTAL's coordinator then leverages command annotations to group subgraphs into one of three types with different fault-recovery semantics (Fig. 3, A1): (1) `main`, which contains the AST region previously deemed as "unsafe" for distribution and is executed on a node containing the authoritative shell state and broader environment—typically, the client node from which the computation is initiated, (2) `regular`, which does not include an aggregator vertex, and (3) `merger`, which includes an aggregator vertex, (*e.g.*, `sort -m`), responsible for merging the outputs of multiple upstream subgraphs. It then instruments every inter-subgraph edge with lightweight communicative primitives (Fig. 3, A2) that record delivered byte offsets and enforce exactly-once semantics over ad-hoc UNIX pipes. Next, a per-subgraph, heuristic-based component (Fig. 3, A3) decides whether to persist each subgraph's outputs locally, balancing recovery speed against fault-free overhead. Once prepared, the coordinator schedules subgraphs across executor nodes and relies on progress and health monitors to track execution progress and detect faults (Fig. 3, A4-6). On a node fault, it identifies and re-executes only the incomplete downstream subgraphs, ensuring both correctness and efficiency in recovery. Executors (Fig. 3, B1-4) receive their assigned subgraphs, reconstruct them into shell scripts, and run them in a tight, non-blocking event loop that maximizes CPU utilization without oversubscription.

**Results**: On a 30-node Cloudlab cluster (§7), FRACTAL executes Fig. 2's script in 220s (speedup: 40×). Upon fault at 50% of the execution, FRACTAL executes only the necessary subgraphs—outputting correct results across all local and HDFS files in 330s (27.1×).

## 4 System Design

This section presents FRACTAL's fault recovery design and core components that drive the execution and recovery.

### 4.1 Fault Recovery in FRACTAL

When the health monitor alerts the coordinator about a node fault, the rescheduling of necessary subgraphs occurs in five steps where FRACTAL (1) identifies all incomplete subgraphs assigned to the crashed node and, by querying progress monitoring, their dependencies; (2) sends `kill` requests to subgraphs that cannot be used in the new execution plan; (3) updates the progress monitor according to the new execution plan; (4) identifies subgraphs that no longer need to be re-executed because their results are persisted; (5) distributes the optimized set of subgraphs based on the new execution plan.

Recovery strategies differ for `merger` and `regular` faults. When a `merger` fails, FRACTAL spawns a new merger and reschedules incomplete upstreams to stream to it, while completed ones send their persisted output. When a `regular` fails, FRACTAL restarts the subgraph elsewhere and tells its downstream merger to resume reading from the last byte, so the merger itself is not rerun.

While the new execution plan is being prepared, the scheduler may receive new dataflow graphs to distribute. To avoid concurrent modifications to the progress monitor and further complications, crash handling and scheduling are performed under locks and are mutually exclusive.

When a loop is unrolled into parallel subgraphs, FRACTAL tracks read-write and write-write dependencies between iterations to establish execution order and isolation. If an iteration's corresponding subgraph fails, the scheduler applies the standard five-step crash-handling procedure only to that iteration and any upstream dependencies, while independent iterations are neither reissued nor re-executed. Completed iterations either reuse persisted outputs or are simply skipped, ensuring faults in one iteration would not force recomputation of its peers unless necessary.

## 4.2 FRACTAL Components

**DFG augmentation**: First, FRACTAL employs PaSh-JIT [34][1], a just-in-time compiler designed to parse POSIX shell scripts into an abstract-syntax tree and emits a command-level DFG. For example, it transforms Fig. 2 into data-parallel `sort` commands and an aggregating `sort -m`. Before scheduling, FRACTAL augments each DFG subgraph with remote pipes, a distributed, exactly-once replacement for the familiar Unix pipes, to track execution progress and ensure correctness during fault recovery. For instance, in Fig. 3 (A4), remote pipes are added at the boundary between the `merger` and `regular` subgraphs and between the `merger` and `main` subgraphs. The scheduler then assigns each subgraph to an executor node, replaces original inter-subgraph edges with these remote pipes, and updates the progress-monitor metadata.

**Progress monitor**: The progress monitor maintains all metadata needed for fault recovery: subgraph-to-node assignments, completion events, and inter-subgraph dependencies. Upon completing a send or receive operation, each subgraph emits a 17-byte completion event to the progress monitor, containing its serialized edge ID and a one-byte flag indicating whether it sent or received. The scheduler then uses these compact messages to determine exactly which subgraphs must be re-executed after a fault. When persistence is enabled, the monitor also tracks file locations so that already-persisted subgraphs are not re-executed after a fault.

The discovery service is a submodule for subgraphs to dynamically locate peers. It acts as a barrier between writers and readers, blocking one until the other registers and initiates data transfer. The service also records persisted-file locations under dynamic persistence, so cached subgraphs are not rerun after fault.

**Remote pipe**: Efficient communication enables precise recovery, as lost or duplicated bytes break correctness and wastes work. Remote Pipes provide unidirectional channels between writers and readers, identified by one edge ID. The reader polls until its matching writer registers before streaming data. Remote pipes run in transient (sockets) or persistent (files) mode, decided by the dynamic persistence switch. If persistence is disabled, the writer opens a socket and registers its endpoint for the reader to resolve; if enabled, the writer writes to a file and exposes its path for retrieving the data during potential re-executions.

Detecting and handling faults is crucial for the remote pipe. If a connection is lost, the reader periodically queries the discovery service. When a new address is found after the upstream is rescheduled, a new connection is made. The reader consumes the stream in buffered chunks and forwards data downstream when the complete data chunk is ready. Since the reader knows how many bytes it has already forwarded downstream, it can discard duplicates and maintain a correct, non-repetitive data stream. This prevents duplicate output from side-effectful operators (*e.g.*, the `append` operator shown in Fig. 2) during re-executions.

**Dynamic output persistence**: Upon node fault, any incomplete subgraph and its upstream dependencies must be re-executed, which can be costly with expensive upstream tasks. To reduce this overhead, FRACTAL can persist the outputs of upstream subgraphs so that reassigned tasks can read cached results instead of recomputing them. However, during fault-free execution, such local writes incur costs that vary with node hardware (*e.g.*, HDD vs. NVMe).

To strike a balance, FRACTAL employs a heuristic-driven dynamic persistence policy to make per-subgraph decisions based on static cluster profiling (*e.g.*, SSD sizes) and runtime workload characteristics (*e.g.*, commands and inputs). Inputs confined to a single DFS block—the smallest fixed-size unit of data replicated across machines—run data-locally; distributing them and persisting their output adds network and replication I/O but no recovery benefit: node fault still loses the cached outputs. Therefore, FRACTAL tags such subgraphs created at DFS block boundaries with no downstream dependents as *singular* subgraphs and skip persistence. FRACTAL

[1]Any compiler that produces an equivalent DFG schema can be plugged in without changes to FRACTAL.

makes per-subgraph decisions: it disables output persistence for *singular* subgraphs and selectively enables it for others based on cost heuristics.

When persisting, the writer appends its output to a temp file, and a per-node file transmitter streams it downstream. The reader still receives data on the fly while each byte is written exactly once, avoiding the extra CPU, memory, and PCIe traffic incurred by alternative "tee-to-disk" designs.

**Health monitor**: Modern DFS typically exposes per-worker heart-beats and block-location metadata for querying liveness. FRACTAL's prototype builds on HDFS [58], a widely deployed reference implementation, and can be extended to any other DFS that expose a similar interface.

The health monitor polls the HDFS namenode's JMX endpoint for each data node's *lastContact* heartbeat timestamp. Nodes whose *lastContact* exceeds a configurable threshold are flagged offline. This threshold involves a balance between false positives and false negatives. A small threshold results in frequent false positives, where temporary network slowdowns are mistaken for faults, triggering costly fault recovery. Conversely, a high threshold causes the system to wait unnecessarily for outputs from faulty nodes. Therefore, the threshold is designed to be configurable. The default value of 10 seconds is selected arbitrarily to ensure it does not dominate the execution time during evaluation (§7), while allowing a substantial portion of the execution to complete before initiating fault recovery mechanisms. This liveness information drives the scheduler's fault recovery decisions, triggering re-executions of affected subgraphs on healthy nodes.

Relying on HDFS heartbeats is an intentional choice to avoid scenarios where FRACTAL nodes appear to be available but HDFS nodes are not, or vice versa.

**Executor runtime**: The executor runtime receives serialized subgraphs from the coordinator and deserializes them into shell scripts. It then stages deserialized scripts and their metadata in a temporary directory for execution.

Every 0.1s, it performs three actions: (1) reclaims completed tasks—removing them from the active pool and recording timing and debug metadata; (2) applies pending `kill` requests from the coordinator by dropping targeted events from the queue; and (3) launches queued subgraphs up to the configured concurrency limit by spawning new processes.

Additionally, the executor runtime also manages environment setup and teardown (e.g., terminating remnants of rescheduled subgraphs to avoid duplicated executions), collects timing and diagnostic metadata, and enables controlled fault injection during evaluation.

**Command annotations**: Previous systems [34, 51, 54, 70] use command annotations to expose parallelisation opportunities. FRACTAL inherits PaSh-JIT's JSON-based annotation catalogue (see Appendix A for example annotations). Commands that depend on (*e.g.*, `ls`) or modify (*e.g.*, `rm`) node-local states, are annotated as side-effectful and therefore pinned to the client-side `main` subgraph to preserve sequential semantics. These annotations enable FRACTAL to distinguish safely re-executable regions (*e.g.*, pure data transformations) from non-re-executable ones (*e.g.*, side-effectful or non-deterministic operations), ensuring only subgraphs containing all safe commands are re-executed on fault. Regions cannot be safely re-executed are offloaded to be part of the main subgraph, which is executed on the client node. The default catalogue already covers 89 common Unix commands, so most scripts require no additional effort. Adding a new entry for a customized command is a single JSON file ($\approx 10$ LOC). When a developer lacks the expertise to annotate a custom command, FRACTAL conservatively treats it as unsafe for fault-tolerant distribution and therefore executes that region on the client node. Such contributions grow a shared annotation repository for shell commands [40, 54], akin to TypeScript's crowdsourced *DefinitelyTyped* [1, 55, 69].

## 5 Optimizations

This section presents targeted optimizations to FRACTAL's critical-path components, reducing control-plane overhead and addressing implementation-specific challenges.

**Event-driven architecture**: The event loop in the executor runtime (§4.2) is among FRACTAL's most performance-sensitive components. To eliminate synchronization overhead, it relies solely on atomic operations (*e.g.*, integer assignments and list append/pop) instead of locks. Completion events are kept to 17 bytes each (edge ID plus direction flag) to minimize messaging overhead. The loop polls every 0.1s to balance kill-signal responsiveness against CPU utilization, and its concurrency level, the maximum number of subgraphs launched in parallel, is configurable per node, defaulting to the CPU core count to match hardware capacity.

**Buffered I/O**: To mark the end of an remote pipe stream, the writer appends an 8-byte EOF token. However, detecting and removing this sentinel on-the-fly is challenging because the reader cannot buffer the entire stream or perform full-stream scans. To address these challenges, the reader employs a buffered I/O strategy with several optimizations. It first allocates a configurable buffer, typically 4096 bytes, and ensures that at least 8 bytes are initially read——this is always possible because the presence of the EOF token guarantees at least 8 bytes of data. After this initial setup, the reader enters a loop that (1) performs another read to fill the buffer following the initial 8-byte segment; (2) sends the buffer's contents, except for the final 8 bytes, downstream; (3) checks whether these last 8 bytes match the EOF token and, if they do, stops reading; and (4) moves the last 8 bytes to the start of the buffer, ready for the next iteration. This approach reduces overhead to at most an 8-byte copy for each iteration without generating unnecessary garbage, offering a significant improvement over simpler, linear parsing methods.

**Batched scheduling**: If a script's input is relatively large or consists of many smaller files, FRACTAL may generate an excessive number of subgraphs to schedule, track, and execute. In such cases, distributing subgraphs can become more time-consuming than executing them. To address this issue, FRACTAL collects and batches all subgraphs with identical targets into a single request and sends these batches asynchronously to all cluster members. Such batching becomes increasingly important as the cluster size grows.

## 6 Fault Injection

To aid parameter selection and recovery characterization, FRACTAL's fault-injection subsystem, available as a command-line tool called **`frac`**, allows injecting runtime fail-stop and fail-restart faults in large-scale distributed deployments. The **`frac`** subsystem is agnostic to deployment and component internals, and has been used to inject faults to FRACTAL with various deployment environments and fault conditions during evaluation. Contrary to manual killing, **`frac`** offers automation, operates at byte-level and millisecond-level precision, can be driven by key events, allows automated restarts—and, by operating at the process-tree level, offers significant performance improvements over complete node shutdown, accelerating parameter selection and recovery characterization.

**Hard faults**: Manually shutting down compute nodes running the executor processes, termed *hard fault*, ensures they end up offline by issuing commands to the host environment. Unfortunately, hard faults are hard to automate at large-scale experiments. Existing VM shutdown tools are not ideal because the aforementioned experiments require non-graceful shutdowns, and verifying that nodes have truly come back up requires custom Docker-level health checks (*e.g.*, polling until each block reaches its target replication factor).

Hard faults additionally do not support fine-grained control over the timing of fault events, crucial for precisely characterizing a system's recovery behavior. This limitation is less problematic for systems with balanced execution progress across nodes. For example, the mapper and reducer of AHS are load-balanced across the cluster and thus a fault injection will hit different nodes at roughly the same execution progress. However, FRACTAL sees imbalanced progression across `regular` and `merger` nodes, thus requiring and benefiting from improved precision in fault injection.

**Soft faults**: The **`frac`** tool offers two *soft faults* modes. Data-plane mode injects a fault token into the stream that matches one node's wrapper. The token travels through the entire DFG, propagated downstream by remote pipes, and is copied by DFG splitters, reaching every parallel branch. Wrappers forward the token unchanged, skipping their wrapped command, except the one on the target subgraph's ingress edge; that wrapper terminates all processes in its subgraph. This mode offers fine-grained byte-level precision for determining the exact point at which to inject a node fault, when the precise fault conditions hinge on specific elements of the data stream.

A control-plane mode sends a special token directly to the node responsible either at a specific time point or by the trigger of a specific event. Additional automation collects baseline execution times about the various jobs on each node. In a subsequent run, the coordinator injects the fault at a configurable time or percentage of a node's fault-free execution time. Focusing on the completion percentage of individual nodes is important in cases where the execution is not balanced—for example, 50% end-to-end job completion does not translate to 50% of AHS's map or FRACTAL's regular node execution, as these nodes typically consume a minority of the runtime. Once it receives a message from the coordinator, the fault terminates its HDFS datanode process and kills the corresponding process. Control-plane soft faults offer coarser-grained precision for determining the poitn at which to inject a fault, often incorporating higher-level goals such as completion percentages.

**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 |

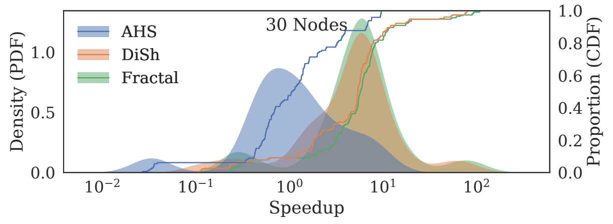


**Fig. 4: Fault-free performance summary.** Summary of AHS, DISH, and FRACTAL 30-node fault-free speedups over Bash across all benchmarks: FRACTAL is comparable to DISH and significantly faster than AHS.

## 7 Evaluation

This section characterizes FRACTAL's performance under fail-free and fault-induced executions.

**Baselines**: We compare FRACTAL with (1) Bash [65], a standard single-node shell interpreter that establishes sequential performance; (2) Apache Hadoop Streaming [26], a

MapReduce-backed system that offers fault-tolerant execution of arbitrary binaries, including black-box Unix commands, providing a production-grade reference for recovery time; and (3) DISH [51], a state-of-the-art system for automatic shell-script distribution that lacks fault tolerance, representing the best published speedups in the fault-free regime. All three run the original scripts unchanged, except that AHS requires the usual mapper / reducer wrappers.

**Benchmarks**: We used five real-world benchmark sets (Tab. 2) from the Koala benchmark suite [39][2], totaling 77 scripts and 547 lines of shell code (LoC) excluding empty lines and comments. `Classics` [2, 3, 33, 46, 64] and `Unix50` [4, 38] comprise scripts that extensively invoke UNIX and Linux built-in commands. `NLP` [7] features scripts from a tutorial focused on developing natural language processing programs using UNIX and Linux utilities. `Analytics` [68, 72] features data-processing scripts, including actual telemetry data from mass-transit schedules during a large metropolitan area's COVID-19 response and multi-year temperature data across the US. `Automation` [54, 57, 59] features scripts for processing, transforming, and compressing video and audio files, typical system administration and network traffic analyses over log files, and aliases for encrypting and compressing files.

**Hardware & software setup**: Experiments were conducted on two clusters: (1) 30 Cloudlab m510 nodes, each with 8-core Intel Xeon D-1548 CPU at 2.0 GHz, 64GB RAM, 256 GB NVMe, and 10-Gb connection; (2) 4 on-premise Raspberry Pi-5 nodes, each with a 4-core Arm Cortex A76 CPU at 2.4 GHz, 8GB RAM, 1TB SSD, and 1-Gb connection. The Pi-5 cluster results confirm FRACTAL's benefits extend to both datacenter servers and resource-constrained edge devices, demonstrating its portability.

To improve reproducibility and ease deployment, we use Ubuntu 22.04-based Docker Swarm images on both 4 and 30 node clusters. We used Bash v5.1.16, Apache Hadoop v3.4.0, Python v3.10.12, and Go v1.22.2.

FRACTAL is developed on top of the PaSh-JIT compiler [34], which includes a Python re-implementation of libdash [23] a POSIX-compliant shell-script parser. FRACTAL adds 2K lines of Python (scheduler, monitors, executor runtime), 1.1K lines of Go (remote pipe and services), and 0.73K lines of shell script. An additional 389 lines of Python and 4.1K lines of shell scripts comprise the `frac` tool.

**Correctness**: The results of over 100 repetitions across several dozen distributed deployments and fault scenarios, over 70 scripts, and over 200GB of inputs are identical to those of the sequential script execution, offering significant confidence about FRACTAL's correct execution and recovery.

[2]These benchmarks were originally introduced in PASH and DISH [34, 51, 70] and later incorporated into the Koala benchmark suite.

**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 |

## 7.1 Fault-Free Execution

This section characterizes the speedup of FRACTAL over Bash against DISH and AHS (Fig. 4).

**Experiments**: We execute all benchmarks on Bash, DISH, AHS, and FRACTAL on both clusters without injecting any faults. While Bash, DISH, and FRACTAL execute all shell scripts without modifications, AHS requires modifications. Not all shell scripts are expressible in AHS; those that are (Tab. 2, col. AHS) are used to compare AHS with FRACTAL.

**Results**: Fig. 5 shows the speedup of the three systems over Bash on the two clusters (key comparisons in Tab. 3). On the 30-node cluster, FRACTAL achieves an average speedup of 9.64× (max: 107.84×) compared to 8.2× (max: 78.35×) for DISH and 1.99× (max: 9.48×) for AHS. On the 4-node cluster, FRACTAL achieves an average speedup of 5.93× (max: 18.55×), compared to DISH's 5.88× (max: 19.04×) and AHS's 1.27× (max: 6.94×).

Excluding the Unix50 and NLP benchmarks, which are not well-suited for scaling across large clusters, FRACTAL achieves an average speedup of 4.66× on a 4-node cluster and 21.90× on a 30-node cluster.

Section 3's log-analysis script (Fig. 2), part of Automation, processes 30GB in 2140s on Pi-5 and 1524s on m510. FRACTAL brings it to 436s (4.90×) on the 4-node Pi-5 cluster and 484s (3.15×) on the 30-node Cloudlab cluster.

**Discussion**: FRACTAL is almost always faster than Bash, but the exact speedup achieved depends largely on the parallelization characteristics of each script. Scripts whose `regular` subgraphs consist of filters (*e.g.*, `grep`) or folds (*e.g.*, `wc`) perform better, as they reduce the runtime fraction used for I/O or the sequential `merger`. Conversely, scripts that do not filter as much or spend more time merging results experience lower speedups. Short-running scripts such as Unix50's `4.sh` and `34.sh` experience slowdowns, as their runtime is dominated by near-instant `heads`—but still remain within 1s.

FRACTAL performs better than AHS because it exploits pipeline, data, and task parallelism for the entire pipeline. By contrast, AHS workflows often comprise multiple `map` and `reduce` stages, each of which must complete before the next begins, leaving parallelism opportunities unexploited.

FRACTAL sometimes outperforms DISH, attributing to FRACTAL's low-overhead output persistence design (§4.2),

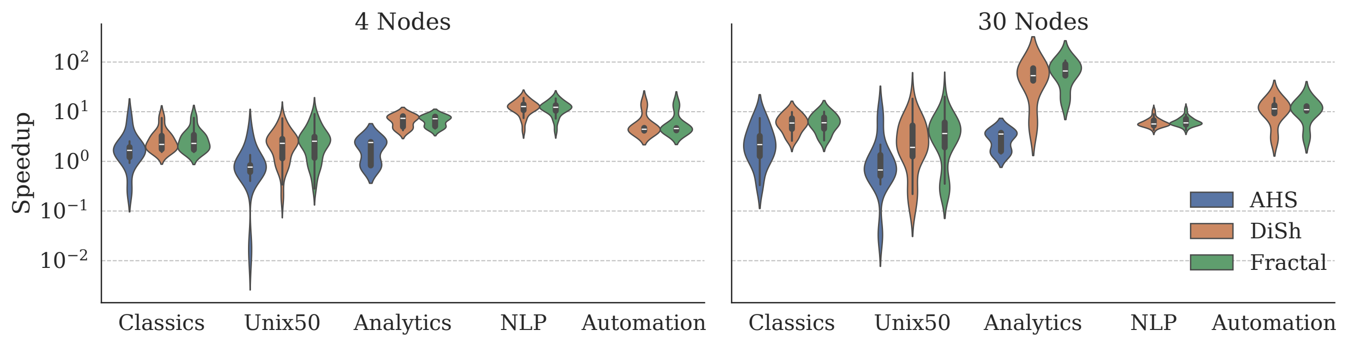


**Fig. 5: Fault-free performance comparison (*Cf.*§7.1).** Comparison between fault-free execution speedups of AHS, DISH, and FRACTAL, relative to single-node Bash, on the 4-node Pi-5 cluster (left) and the 30-node Cloudlab cluster (right).

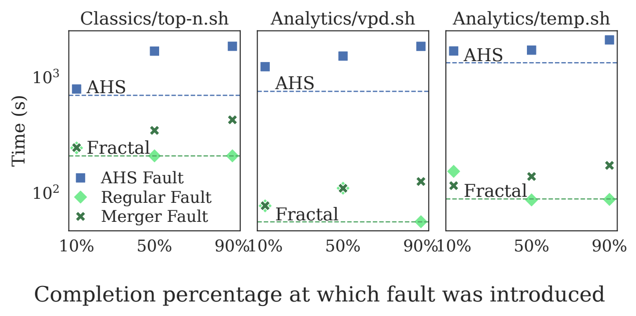


**Fig. 6: Recovery comparison (*Cf.*§7.2).** Comparison between the FRACTAL and AHS recovery times for 3 representative scripts (left, mid, right), with faults introduced at 10%, 50%, and 90% of the execution—and without faults (dashed lines).

control-plane optimizations (§5), and minimal fault-tolerance overheads in fault-free execution paths. First, with persistence on, FRACTAL uses a local file as a proxy for the socket send, avoiding the extra CPU, memory, and PCIe traffic incurred by alternative "tee-to-disk" designs. Unlike DISH's blocking TCP buffers, FRACTAL uses effectively unbounded buffers, so it can pre-compute larger execution units, advantageous in workloads with complex, interdependent subgraphs where DISH's buffer constraints become bottlenecks. Second, FRACTAL's control-plane optimizations (*e.g.*, asynchronous batching) scales well in fault-free executions, explaining the larger gains on 30 nodes. Lastly, even in fault-free runs, FRACTAL's fault-tolerance overhead is minimal: each executor in `Classics` only adds 136B over the network and writes 1MB to disk. FRACTAL's extra sequential write becomes more visible when (1) a script forms wide fan-in pipelines whose many internal edges are all persisted, (2) those edges carry gigabytes of data yet do almost no computation per byte, and (3) the job runs on a very small cluster, so control-plane optimization gains are negligible.

The smaller speed-up on the larger cluster with the example log-analysis script (Fig. 2) is because the job is I/O-bound rather than CPU-bound. Its critical path consists of (1) streaming full log files over the network instead of reading from the local page cache and (2) the single-threaded `merger`, whose `sort` now merges eight remote partitions. On the 4-node Pi-5 cluster the data remain largely local and only four partitions need merging, so the relative gain is higher.

## 7.2 Performance of Fault Recovery

We characterize FRACTAL's fault recovery with one experiment comparing FRACTAL with AHS failing at various stages and another characterizing FRACTAL under more scenarios.

**Experiments**: The first experiment assesses the time it takes FRACTAL and AHS to recover and successfully complete the job on the 4-node deployment. We introduce faults at approximately 10%, 50%, and 90% of the baseline execution time: at 10%, AHS executes mappers and FRACTAL executes `regular` subgraphs; at 90%, AHS executes reducers and FRACTAL executes `merger` subgraphs. In separate runs, we inject faults into an arbitrary AHS node, a base-case `regular` node, and a worst-case `merger` node. Combining all these configurations with hard faults takes prohibitive manual effort, so we focus on three of the collected benchmarks here.[3]

The second experiment zooms into FRACTAL's fault recovery under different fault scenarios on both clusters. It employs soft faults using **`frac`**, and all benchmarks except NLP and Unix50 as they contain many short-running scripts.

**Results**: Fig. 6 summarizes the first experiment. The x-axis shows different completion percentages and the y-axis shows the time it takes AHS and FRACTAL (both `regular` and `merger` nodes) to recover. For context, dashed lines (constant across the x-axis) represent the fault-free executions for AHS (avg: 937.6s) and FRACTAL (avg: 118.9s). Under `regular` faults, it takes FRACTAL 160s (134.5% vs. fault-free), 136.5s (114.8%), and 118.8s (100.1%) to recover for

[3]Total: 3 completion percents × 3 system configs (AHS, `regular`, `merger`) × 2 fault modes × 5 repetitions × 3 benchmarks = 270 experiments (about a week of manual effort) instead of 6,930 experiments.

**Tab. 4:** FRACTAL's speedup over AHS for different fault conditions and recovery scenarios. Format: avg (min–max).

| | **Regular Recovery** | **Merger Recovery** |
|---|---|---|
| Fail at 10% | 7.8× (3.2–16.0×) | 8.5× (3.2–15.9×) |
| Fail at 50% | 12.1× (8.0–19.7×) | 8.3× (4.8–13.9×) |
| Fail at 90% | 16.4× (8.9–32.9×) | 8.0× (4.3–14.3×) |

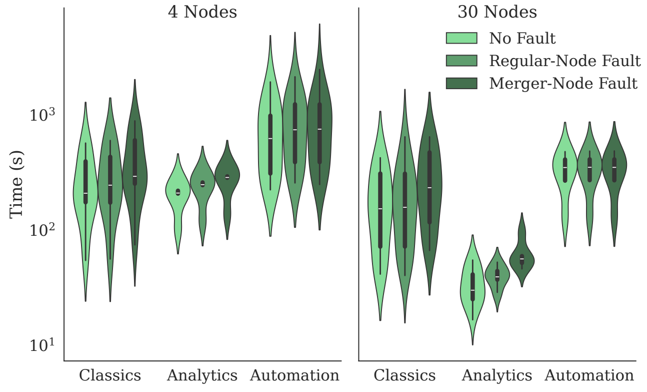


**Fig. 7: Recovery comparison (soft faults) (*Cf.*§7.2).** FRACTAL execution times for three benchmark sets (Classics, Analytics, Automation) with no faults, `regular` faults, and `merger` faults on a 4-node Pi-5 cluster (left) and a 30-node Cloudlab cluster (right).

each execution percentile; under `merger` faults, these become 147.7s (124.2% vs. fault-free), 200.2s (168.3%), and 244.4s (205.6%) respectively. For the same percentiles, it takes AHS 1,248.8s (133.2% vs. AHS fault-free, 780.9% vs. `regular`, 845.5% vs. `merger`), 1,655.8s (176.6% vs. AHS fault-free, 1,213.2% vs. `regular`, 727.1% vs. `merger`), and 1,953.4s (208.3%, 1,644.3%, 799.1%). Tab. 4 summarizes the comparison between AHS and FRACTAL.

Fig. 7 summarizes the second experiment, showing execution times for fault-free, `regular` recovery, and `merger` recovery. Benchmarks with fewer parallel pipelines (*e.g.*, Classics and Analytics) take 20.3–32.1% longer to recover from `merger` (209.4–344.8s) than `regular` node faults (150.4–277.6s). For other benchmarks, there is not a significant difference between the recovery of different nodes (335.3–845.0s for `regular` vs. 335.7–856.5s for `merger`).

**Discussion**: Overall, the first experiment shows that FRACTAL recovers at a fraction (6.08–12.8%) of AHS's recovery time. The speedup stems from selectively re-executing only affected tasks while preserving parallelization benefits (§7.1) during recovery.

Recovery time increases for faults that occur later in execution (Fig. 6) because more upstream subgraphs must be re-run. FRACTAL's `regular` faults are an exception to this pattern because the `regular` subgraphs have already completed by the time the node fails. This non-interference is important in practice: `regular` nodes make up most of distributed executions, so a random infrastructure fault is most likely to affect a `regular` node.

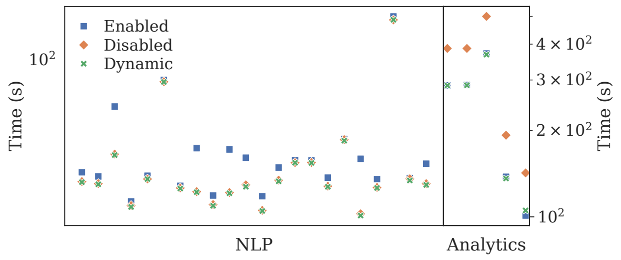


**Fig. 8: Microbenchmark: dynamic persistence (*Cf.*§7.3).** Fault-free NLP benchmark (left) and fault-injected Analytics benchmark (right) with dynamic persistence enabled, disabled, and set dynamically by FRACTAL's heuristics.

As this experiment compares *hard* faults introduced manually and *soft* faults injected by **`frac`**, it confirms that the two modes result in identical executions across 270 experiments—but **`frac`** completes experiments at about 2–5% of the hard-fault time, and without the mental overhead of keeping manual track of various experiment timepoints.

Diving into various types of recovery (Fig. 7) indicates that the pipeline-to-node ratio of pipelines is correlated with `merger`-to-`regular` node recovery performance. This observation is intuitive for two reasons. First, benchmarks that rely on a single pipeline (*e.g.*, `Classics` and `Analytics`) experience longer recovery times from `merger` faults than from `regular` faults, since `regular` faults involve re-executing fewer subgraphs. Second, benchmarks with many pipelines (*e.g.*, `Automation`) are indifferent to `merger` and `regular` recovery times: having a larger number of `merger` subgraphs distributes the workload evenly, effectively making every node a `merger` node and neutralizing the impact type.

## 7.3 Microbenchmark: Dynamic Persistence

This experiment evaluates dynamic subgraph-output persistence and how it strikes a balance between fault recovery efficiencies and overhead during fault-free execution.

**Experiments**: We explore output-persistence trade-offs using two benchmark sets under different fault conditions and with both persistence options (enabled and disabled): `NLP (no faults)` features many parallel pipelines, each using a small input (<128MB); `Analytics (merger faults)` features long-running regular (upstream) subgraphs. The pair captures the two extremes our heuristic must handle: `NLP` stresses fault-free overhead where persistence should be off, whereas a merger-node crash in `Analytics` stresses recovery time where persistence should be on.

**Results**: Fig. 8 compares three output-persistence options. For fault-free `NLP`, enabling persistence results in 21.0% overhead on average; for fault-injected `Analytics`, disabling persistence results in 38.7% overhead on average. At runtime, FRACTAL's dynamic persistence switch selects the optimal option for both cases, balancing performance and fault tolerance effectively.

**Discussion**: Enabling output persistence for short-running workloads adds considerable overhead yet offers little benefit. `Regular` and `merger` subgraphs are typically co-located, so a fault makes both unavailable and forces re-execution regardless of persistence. Failed long-running scripts see significant benefits from output persistence, as they avoid significant upstream re-execution. FRACTAL first-order workload heuristics—*e.g.*, size of DFG graphs, input sizes—decide whether to enable persistence for the the benchmarks.

## 8 Related Work

FRACTAL is related to a large body of work in distributed shells and shell-related utilities, distributed computing frameworks, and language-based distributed systems.

**Distributed shells and utilities**: Several command-line job-scheduling tools allow distributing workloads on Unix systems—*e.g.*, `qsub` for the Sun Grid Engine [20] and `parallel` for GNU Parallel [62]—but their invocation requires careful manual orchestration and does not come with fault tolerance built-in. Slurm [73], a workload manager for distributing batch jobs across computing clusters, provides periodic check-pointing for later resumption using DMTCP [35]. This mechanism focuses only on recovering Slurm-visible state, does not support complex commands, and fails to account for thorny shell semantics such as `append` (§3).

Shells like Rc [13], Dgsh [60], and gsh [45] offer scalable, often non-linear and acyclic, extensions to Unix pipelines but require manual rewriting and do not tolerate faults.

Recent systems such as PaSh [34, 70], POSH [54], and DISH [51] offer automated parallelization and distribution of shell programs. Similar to FRACTAL, they automatically transform scripts to internal representations to be parallelized and distributed; however, they are not fault-tolerant.

**Distributed computing frameworks**: FRACTAL combines elements from distributed batching and streaming systems [11, 47, 49, 50, 53, 61, 71, 74]. These systems offer the ability to tolerate faults, often by tracking lineage similarly to FRACTAL [47, 74], but require their users to (re-)implement their programs using the abstractions these systems provide. These systems do not support the black-box nature, runtime expansion behaviors, and arbitrary side effects pervasive in the commands typically present in shell programs.

Hadoop Streaming [26] and Dryad Nebula [30] are unusual in that they allow the use of black-box components such as Unix commands. However, they do not target the semantics of the shell and thus require users to manually port their shell programs—often facing the difficulty or inability to express entire classes of shell programs as FRACTAL's evaluation confirms. FRACTAL provides automated scale-out of unmodified shell scripts, supports the shell's dynamic-expansion features, and offers efficient fault tolerance by tracking lineage.

**Other cloud offerings**: Prior work on VM- and container-level replication has used check-pointing [9, 10, 14, 36, 44] to tolerate faults. Contrary to FRACTAL, these approaches leverage logging, require infrastructure support, and lack a logical understanding of the workload.

Serverless platforms have started introducing stateful operations [32, 43, 75] and thus fault tolerance, through a combination of logging and check-pointing. Different from FRACTAL, these systems do not support shell scripts or arbitrary black box commands—users need to (re)write their scripts in the abstractions provided by these systems to see benefits.

The `gg` system [17] supports scaling black box commands to serverless functions. In contrast to FRACTAL, `gg` does not support pipeline parallelism and attempts full executions via a re-try mechanism when faults occur.

## 9 Limitations

FRACTAL's fault model focuses on high-impact faults that are especially important for the shell, and so does not address orthogonal types of faults. First, FRACTAL assumes a highly available coordinator, which could be achieved by using any consensus protocol such as Raft [52] to replicate its state (*e.g.*, progress log, discovery metadata). Second, FRACTAL does not yet distribute or recover side-effectful commands that mutate external state (*e.g.*, `mv`, `curl`, database writes). One way to address that would be to use a sandbox mechanism, such as TRY [67], which can record and roll back file-system changes and therefore enable capturing and replaying side-effectful subgraphs.

Finally, while the single-coordinator design scales comfortably to our 30-node experiments, larger container-dense deployments may require sharding or a hierarchical scheduler. Exploring this control-plane scaling is left for future work.

## 10 Conclusion

Transparent fault tolerance is a *sine qua non* for scalable shell-script distribution: without it, unmodified POSIX scripts cannot reliably handle the black-box binaries, ad-hoc pipes, non-idempotent side effects, and dynamic control flow that characterize real-world workflows.

FRACTAL is the first system that offers fault-tolerant shell-script distribution by separating recoverable from side-effectful regions. It performs lightweight instrumentation to record byte-level progress and enforce exactly-once semantics. By employing precise dependency and progress tracking at the subgraph level, it offer sound and efficient fault recovery.

## Acknowledgements

We thank the NSDI'26 reviewers for their feedback; our shepherd, Eric Eide, for his guidance; the NSDI'26 Artifact reviewers for their time; and the Brown CS2952R (Fall'24) participants for their input on several iterations of this paper. This material is based upon research supported by NSF awards CNS-2247687 and CNS-2312346; 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] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *European conference on Object-oriented programming*, pages 428–452. Springer, 2005.

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

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

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

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

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

[17] 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.

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

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

[20] 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.

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

[22] 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.

[23] Michael Greenberg. libdash. https://github.com/mgree/libdash, 2019. [Online; accessed November 22, 2024].

[24] Michael Greenberg and Austin J Blatt. Executable formal semantics for the posix shell. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[25] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix shell programming: the next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 104–111, 2021.

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

[27] Saurav Haloi. *Apache zookeeper essentials*. Packt Publishing Ltd, 2015.

[28] Thomas Haynes and David Noveck. Network file system (nfs) version 4 protocol. Technical report, 2015.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] Gene Kapadia, Jason Ansel, Kapil Arya, Charles Guo, Daniel Maze, Mihir Modi, Cameron Musco, Alexey Lory, and Gene Cooperman. Dmtcp: Distributed multithreaded checkpointing. https://dmtcp.sourceforge.io/, 2024. Accessed: 2024-11-29.

[36] 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.

[37] 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.

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

[39] 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.

[40] Lukas Lazarek, Seong-Heon Jung, Evangelos Lamprou, Zekai Li, Anirudh Narsipur, Eric Zhao, Michael Greenberg, Konstantinos Kallas, Konstantinos Mamouras, and Nikos Vasilakis. From ahead-of-to just-in-time and back again: Static analysis for unix shell programs. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 88–95, 2025.

[41] Haoyuan Li. *Alluxio: A virtual distributed file system*. University of California, Berkeley, 2018.

[42] Georgios Liargkovas, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. Executing shell scripts in the wrong order, correctly. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 103–109, 2023.

[43] 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.

[44] 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.

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

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

[47] 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.

[48] 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*, SOSP '13, pages 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 167–180, 2015.

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

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

[58] 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.

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

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

[61] 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.

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

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

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

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

[66] 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.

[67] 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.

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

[69] TypeScript Authors. TypeScript. https://www.typescriptlang.org/.

[70] 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.

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

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

[73] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[74] 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.

[75] 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.

## A Example Annotation Interface

This appendix shows two example annotations for `rm` and `sort` commands, as discussed in §4.2. Each annotation is a JSON object with a `command` field and a list of `cases` that describe the command's behavior under different `predicates`. For each case, the `inputs` and `outputs` fields specify the command's input and output channels, respectively. The `class` field indicates whether the command is `pure`—non-parallelizable but side-effect-free, `side-effectful`—non-parallelizable and side-effectful, `stateless`—parallelizable without requiring custom aggregation, or `parallelizable_pure`—parallelizable but requiring user-defined aggregation. The `properties` field lists additional properties of the command, such as `commutative` specifying that the command allows splitting and processing data-parallel partial inputs in order-independent batches. A full description and discussion of the annotation can be found in PASH papers [34, 70].

For example, the `rm` command is side-effectful as it deletes files and does not support parallel execution.

```
{
  "command": "rm",
  "cases": [
    {
      "predicate": "default",
      "class": "side-effectful",
      "inputs":  ["stdin"],
      "outputs": ["stdout"]
    }
  ]
}
```

The `sort` command is non-parallelizable when the `-m` option is provided, as it merges pre-sorted inputs. When `-m` is not provided, `sort` is parallelizable and requires aggregation using `sort -m` to ensure correct results.

```
{
    "command": "sort",
    "cases":
    [
        {
            "predicate":
            {
                "operator": "exists",
                "operands": ["-m"]
            },
            "class": "pure",
            "inputs": ["args[:]"],
            "outputs": ["stdout"]
        },
        {
            "predicate": "default",
            "class": "parallelizable_pure",
            "properties": ["commutative"],
            "agg": "sort",
            "inputs": ["stdin"],
            "outputs": ["stdout"],
            "aggregator":
            {
                "name": "sort",
                "options": ["-m"]
            }
        }
    ],
}
```