Incr: Faster Re-execution via Bolt-on Incrementalization

Yizheng Xie* Evangelos Lamprou* Jerry Xia* Nikos Vasilakis
Brown University

Abstract

While most software development is incremental, most execution environments are not: even small program modifications fail to take advantage of prior executions, at worst requiring full re-execution of all computational stages in the modified program. Such full re-execution decelerates software development and debugging, especially in dynamic polyglot environments such as the Unix and Linux shell. This paper presents INCR, a system that accelerates the re-execution of unmodified shell programs by automatically incrementalizing their execution. INCR analyzes and tracks interdependencies to detect and store key intermediate results, reusing them on subsequent re-executions whenever possible. INCR's effect analysis guarantees correct re-execution even for non-idempotent computations, and several static and dynamic optimizations reduce the runtime and storage overheads of incrementalization. Applied to diverse real-world scenarios, INCR accelerates re-execution by an average of 34.2× and a maximum of 373.3×—all while requiring no developer annotations or code modifications and remaining behaviorally indistinguishable from non-incremental execution.

1 Introduction

Nearly all software development is incremental: layers of modifications, additions, replacements, and deletions are applied iteratively to morph a program toward its intended goal. Such incremental development is particularly common today in data science [45, 63, 85], machine learning [17, 37], exploratory computing [30, 68, 80], and interactions with large language models [35, 69]. It is also the standard approach in dynamic, interactive environments such as the Unixshell, which is used for understanding, exploring, and gradually refining software systems, the opaque components comprising them, and the input they operate on [38, 50, 79].

Unfortunately, these environments do not support efficient re-computation when parts of a program change. Such changes, irrespective of their size, fail to take advantage of past executions—requiring a full re-execution of all the computational stages in the modified program. Even small refinements can result in long waiting cycles, especially on large datasets where re-execution might take the majority of the time. As the program grows, the longer it runs—and the slower it becomes to further iterate, discover errors, hand-tune, debug, and eventually ship.

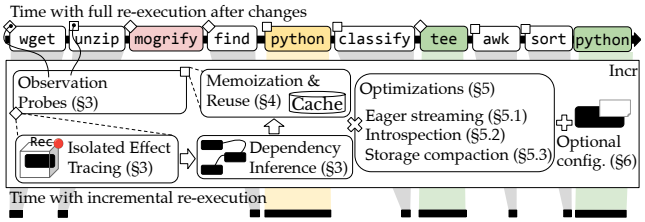

Fig. 1: **INCR overview**. INCR accelerates re-execution in modified shell programs by bolting onto otherwise unmodified execution environments. Upon re-execution, it reuses prior results to avoid redundant computation. Colors indicate types of incrementalization: + additions, − deletions, and ~ in-place modifications.

This paper introduces INCR, a system that accelerates re-execution by bolting incremental re-computation onto unmodified shell programs. Once enabled during development, it trades off a small overhead during the first execution to accelerate subsequent re-executions. INCR operates in two phases: the analysis phase, which collects and analyzes program dependencies; and the incrementalization phase, which extracts information about modifications and, combined with the earlier analysis, accelerates subsequent re-executions of modified programs. INCR supports full POSIX *and* Bash semantics, including modifications to command arguments, flags, data flow, control flow, environment variables, and external resources, ensuring behavioral equivalence even for non-idempotent operations. When ready to ship, INCR can be disabled to avoid unnecessary runtime overheads in production environments.

INCR's analysis and incrementalization phases discover and

*Equal contribution.

exploit process-level data and control dependencies across commands, automatically detecting, analyzing, and storing effects at runtime (§3). In subsequent runs, INCR determines which parts of a shell program are unchanged and safely reuses their outputs (§4). Several additional optimizations, such as eager stream processing, introspection, and storage compaction, lower runtime and space overheads (§5), making incrementalization practical for real-world workloads (§7). While no user input is necessary, INCR can leverage crowdsourced partial annotations for POSIX, GNU Coreutils, and third-party commands from earlier research [48, 74, 86], as well as developer-specified configurations to enable finer-grained incrementalization and eliminate dependency analysis in parts of programs that will not change (§6).

INCR has been applied to 14 real-world scenarios that include debugging, data exploration, interactions with LLMs, and other common tasks (§7). With no developer input, INCR accelerates re-executions by an average of 34.2× and by up to 373.3×. Enabling INCR slows down the first (typically short) execution by an average of 101.50%, mostly due to dependency tracking, reducible to 43.55% using optional developer configurations that disable tracking for stable parts of the program—and further accelerate re-execution by an average of 1.46× and a maximum of 24.40×. INCR's behavior and outputs are indistinguishable from the underlying shell interpreter across all real-world scenarios and 10,279 out of 10,282 (99.9%) tests from the standard Bash test suite, including unusual behaviors rarely seen in practice.

The paper starts by exemplifying INCR on a real-world workload (§2), followed by its key contributions (§3–6):

- Fine-grained dependency tracking via lightweight interposition probes that capture interactions across the filesystem, shell environment, and other external resources (§3).
- Correct incrementalization via memoization of dependencies and effects, including both transient data streams and side effects, and safe reuse of prior effects (§4).
- Diverse runtime optimizations, such as eager stream processing, introspection, and compaction, that make incremental execution practical (§5).
- An optional tuning interface that accepts crowdsourced annotations and developer configurations to enhance, disable, or relax parts of incrementalization (§6).

The paper then characterizes INCR on real-world programs (§7), discusses related work (§8), and concludes (§9).
Availability: INCR is available as MIT-licensed open-source software at:

https://github.com/atlas-brown/incr

2 Applying INCR During Development

Example script: Fig. 2 shows a real shell program used to digitize images of hieroglyphs collected during an archaeological expedition [7]. The script (1) fetches the images collected during the expedition; (2) unpacks the images into a local directory; (3) iterates over the images; (4) segments each image into hieroglyph regions using Meta's Segment Anything Model [52]; (5) applies a hieroglyph classifier to each segment; and (6) outputs the formatted mappings between image filenames and their classified segments to the `db.txt` file.

```
1  wget 'http://inst.edu/dpt/images.zip'
2  IMGS=${IMGS:-images}
3  unzip images.zip -d "$IMGS"
4  mogrify -resize 1024x1024\> "$IMGS"/*        # -(1)
5  for img in "$IMGS"/*; do
5  for img in $(find "$IMGS" -type f); do    # ~(3)
6   python segment.py "$img" |
6   python segment.py "$img" -s 1024x1024 |   # ~(1)
7   while read -r mask; do
8    python classify.py -i "$img" -r "$mask" |
9   done |
10  tee -a classes.txt |                      # +(4)
11  awk '{print "g:", $5}'
11  awk -vi="$img" '{print "g:", $5, $6, i}'  # ~(2)
12 done | sort > db.txt
13 python plot.py classes.txt                 # +(4)
```

Fig. 2: **Hieroglyph classification script**. A script that segments, classifies, and visualizes images of hieroglyphs. Highlighted lines indicate modifications made during development: `+` additions, `-` deletions, and `~` in-place modifications. Numbers denote modification order; equal numbers belong in the same iteration.

Developing this script involved several modifications [4], four of which are highlighted in Fig. 2: removing a redundant image resizing step (ln.4, ln.6, #1), changing `awk` to include image paths in the output (ln.11, #2), making the outer loop process images in all subfolders (ln.5, #3), and plotting the classification results (ln.10, ln.13, #4).
Problem: Re-executing the entire script after each modification takes up to 15 minutes, dominated by the segmentation and classification stages. Incrementalization—re-executing only the parts of the script necessary to reflect each modification—significantly accelerates re-execution.

To achieve incrementalization, a developer could *manually* insert `tee` commands and `if` guards (lns. 5–12) to memoize intermediate results and reorganize the script to reuse those intermediate files. However, this approach requires significant effort and can introduce subtle errors, as manual partial re-execution may use stale intermediate results. Furthermore, this approach does not help with change #3, which expands the input image set, and is cumbersome to apply with changes #1 and #3, which modify the program in non-trivial ways.
Key challenges: There are several challenges in automating

the incrementalization of shell programs. First, Fig. 2's script contains opaque, heterogeneous commands with complex and often implicit interdependencies. For example, `plot.py`, introduced in modification #4, consumes `classes.txt` (produced in ln. 10) and thus depends on the outputs of all prior stages (lns. 5–9). Commands also interact with external resources and the execution environment. For instance, `segment.py` (ln. 6) reads images from the `$IMGS` directory (defined in ln. 2) and depends on several Python packages loaded by the `python` interpreter. In addition, such dependencies are typically implicit and dynamic (*e.g.*, `$IMGS` may be externally set). Therefore, the script's behavior can change with its surrounding context.

Second, these commands generate arbitrary side effects, including modifications to the filesystem, interactions with the network, and transient input/output streams that flow through pipelines and cannot be easily reused across runs. For example, modification #4 alters the pipeline by inserting a `tee` command (ln. 10), which consumes the output stream produced by earlier commands. The `tee` command then emits two effects: one written to disk as `classes.txt`—later consumed by the newly added `plot.py` command (ln. 13)—and another directly streamed to the next command `awk` (ln. 11).

Finally, commands exhibit diverse execution patterns that can make incrementalization via dependency tracing and memoization inefficient. For instance, modification #3 filters irregular files in subdirectories; although this may not change the set of images processed, it introduces a dependency change that may unnecessarily invalidate memoized results. Naïve memoization of command effects can also incur substantial storage overhead, particularly for large artifacts produced by commands such as `unzip` or `segment.py`.

Applying INCR: Developers enable INCR during development to automatically incrementalize scripts and disable it when ready to deploy. This workflow for Fig. 2's script is:

```
$ incr dpt.sh                    # Enabling INCR
$ incr dpt.sh ... incr dpt.sh    # #1 #2 #3 #4
$ ./dpt.sh                       # Ready to ship!
```

To solve the earlier challenges, INCR probes the script's execution at runtime to track command dependencies (§3), memoize their intermediate effects (§4), and reuse them when relevant state is unchanged. It parses the script, inserts *interposition probes* before all command invocations, and executes the transformed script using the underlying shell interpreter. These probes wrap each command and observe their execution: each probe creates an isolated environment for its command to run in, tracks its `stdin` and filesystem dependencies, and memoizes its input/output streams and filesystem effects. During re-execution, each probe compares the current dependencies to those stored and decides locally to re-execute its command if it detects any changes; otherwise, it skips execution and emits its memoized effects. As INCR's probes operate at runtime, they can observe dynamic changes to variables (which may depend on external state) and control flow.

Modification #1 removes the redundant `mogrify` command that resized images before segmentation and makes the `segment.py` command resize images internally instead. Upon re-execution, INCR detects that `segment.py`'s input has changed; INCR thus re-executes `segment.py` normally but detects that its output, containing normalized coordinates, has not changed. Then, the probes surrounding `classify.py` and `awk` detect that their environment, inputs/outputs, and filesystem dependencies are unchanged, so INCR skips their executions and emits their memoized results.

Modification #2 includes two changes to the `awk` command: (1) adding classification confidence (`$6`) to its output; and (2) adding the image path (`i`) to its output. `awk`'s probe detects that its arguments have changed and re-executes it. The downstream probes detect `awk`'s output changes and re-execute their corresponding commands. All prior stages (lns. 5–9) reuse their memoized results.

Modification #3 filters out irregular files in the `$IMGS` directory using `find`. The probe on `segment.py` retrieves and emits memoized results for images it has processed before; otherwise, it executes the command normally.

Finally, modification #4 includes several changes that add a plotting section to visualize the classification results saved by `tee`. On re-execution, INCR skips all unchanged stages before `tee` and executes `tee` on memoized results. Then, INCR detects that `tee` does not alter the input to the `awk` command, and thus continues to reuse memoized results for all subsequent stages—executing only the new `plot.py` command.

Results: INCR improves the original re-execution time from 1h25m to 20m41s, resulting in a 4.1× speedup. Modifications #3 and #4, which include exploration of already-processed data, enjoy 91.2× and 119× speedups, respectively. If annotations were used, INCR would achieve an additional .05× speedup by skipping dependency tracking on `awk` (§6).

3 Dependency Tracking

This section describes how INCR acquires fine-grained information about command dependencies.

3.1 Inserting Interposition Probes

Shell programs feature complex control flow and interaction with the environment, complicating the extraction of command dependencies and their effects. INCR performs all dependency tracking at runtime, following a component-centric approach: it isolates each command's effects and tracks its dependencies individually (Fig. 3). It allows tracking commands on a per-effect basis (§4), enabling far more precise memoization and reuse than probing the script as a monolith.

INCR first parses the shell script using the `libbash` library, which exposes Bash's parsing subsystem as an API. It then walks the program's abstract-syntax tree (AST) and inserts

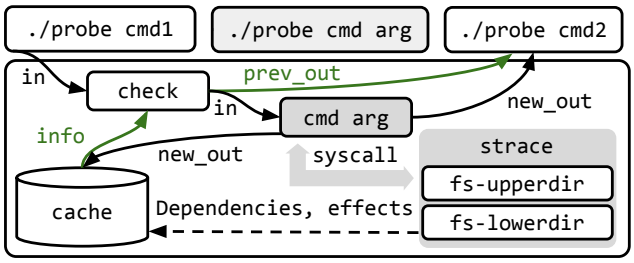

Fig. 3: **Interposition probes**. INCR inserts lightweight interposition probes before each command in a shell program. A command's probe records its dependencies, inputs, outputs, and other effects by tracing its system calls (gray arrows) and memoizing those effects to the cache. On subsequent executions, the probe checks its dependencies against prior runs retrieved from INCR's cache. If they are unchanged, the probe reuses prior results (green arrows); otherwise, it re-executes and re-traces the command (gray arrows).

probes as meta-commands that wrap the original command invocation. For example, INCR transforms the invocation `rm $path` into `incr-probe rm $path`. The probe overwrites the `$0` variable to point to the original command and then invokes it with its original arguments. Each probe is responsible for tracking its command's dependencies (§3.2) and memoizing its effects (§4).

Probes are only placed at trackable commands, which leaves out shell-defined functions, built-in commands, and other components that cannot be resolved inside the shell (*e.g.*, `alias`). To deal with these shell-specific constructs, INCR applies a small set of lightweight syntactic analyses and transformations. In particular, INCR identifies built-in commands (using the closed set given by `compgen -b`), function definitions, and `alias`ed invocations to avoid probe insertion. For other syntactic constructs that create non-observable side effects, INCR lifts them into observable commands. For instance, it converts redirections (`>` and `>>`) into trackable `dd` invocations before inserting probes.

3.2 Detecting Dependencies

Correct incrementalization requires comprehensive tracking of each command's runtime dependencies, including filesystem and shell environment dependencies.

To precisely capture each command's effects on the filesystem, INCR executes them within an isolated environment built on top of an overlay filesystem [16, 73], which provides a private, copy-on-write view of the filesystem. Before each command executes, INCR instantiates a new OverlayFS mount that composes one read-only lower layer (*lowerdirs*) with a writable upper layer (*upperdir*) into a unified merged view. All modifications performed by the command are captured in the upperdir, while the lowerdirs offer transparent read-only access to the underlying filesystem. Upon completion, INCR scans the upperdir to identify modifications made by the command and commits them from the upperdir back to the persistent filesystem. Furthermore, INCR stores the upperdir for reuse during subsequent executions (§4).

Scanning the upperdir only reveals write dependencies. To capture read dependencies, INCR monitors system calls made by each command during its execution. To avoid prohibitive overhead, INCR intercepts only system calls relevant to the dependencies it models—specifically, `fork`, `exec`, and all file-related system calls included in `strace`'s `%file` system set. INCR uses `seccomp-BPF` to filter out irrelevant system calls and reduce context-switching overhead.

INCR also tracks environment dependencies such as environment variables and function declarations. These are pervasive in shell scripts as commands often rely on environment variables for configuration (*e.g.*, `LC_ALL` affects `sort`'s ordering behavior). Before executing each command, INCR captures the current environment variables and function declarations in the shell. If this set differs on re-execution, INCR reruns the command. Environment modifications made during execution are handled automatically as INCR captures environment snapshots on a per-component basis.

Within the execution environment, *false* and *noisy* dependencies may lead to unnecessary re-executions. For instance, INCR cannot determine which environment variables a command depends on as they are accessed at the application level. This forces INCR to conservatively treat all environment variables as dependencies, which may trigger re-executions even when variables that do not impact command behavior change across runs. To mitigate this, INCR implements a distribution-aware filter that discards noisy environment variables. The current INCR implementation is aware of noisy variables inside the Debian and Ubuntu distributions, specifically those related to session management and `tty` settings.

4 Memoization and Reuse

This section explains how INCR memoizes command dependencies and effects at runtime (§4.1) and reuses these memoized results (§4.2) to avoid redundant re-computation.

4.1 Efficient Memoization

INCR stores each command's dependencies and effects collected during effect tracking (§3.2) in a cache directory on the host system. Commands are indexed by their invocation arguments, environment variables, and `stdin` stream hash. During re-execution, INCR indexes into the cache to check if commands can be skipped and to fetch memoized results.

Each command's probe only has visibility into its own read and write dependencies. After recording them, INCR generates a dependency file. INCR tracks read and write dependencies

differently as an optimization, using the insight that components in dynamic runtimes resolve sub-components at runtime, leading to many more read than write dependencies [87]. Write dependencies are tracked using content hashes while read dependencies are tracked using last modification timestamps, which are much cheaper to extract. To check if a write dependency has changed, INCR compares the file's current hash to its stored hash. This allows INCR to skip re-execution if upstream commands modify a file but produce the same output (*e.g.*, `sort` may produce the same output given different inputs), while also distinguishing between same-content overwrites and appends, ensuring correct reuse for non-idempotent writes. To check if a read dependency has changed, INCR checks if the file's timestamp has changed. However, this over-approximates changes when a command reads a file modified upstream. To mitigate this, INCR avoids updating file timestamps when applying memoized writes, ensuring that commands do not detect spurious timestamp changes.

Furthermore, INCR stores each command's transient data streams and filesystem effects alongside its dependency file to memoize its effects. To capture transient data streams, INCR duplicates a command's `stdout` and `stderr` to files within the cache directory at runtime. To capture filesystem effects, INCR directly stores the upperdir generated by OverlayFS during dependency tracking (§3.2).

4.2 Safe Reuse

Correct incremental execution requires detecting when a command's dependencies remain unchanged and applying prior results correctly. During each execution, INCR compares each command's current dependencies against its dependency file. If any differs, INCR re-executes the command and records its new dependencies and effects; otherwise, INCR skips execution and applies its memoized results.

For output streams, INCR directly streams memoized `stdout` and `stderr` to their corresponding file descriptors. For filesystem effects, INCR scans the memoized OverlayFS upperdir that contains a component's post-execution set of filesystem effects. Inside it, created or modified files are represented as regular files, deleted files as *whiteouts* (character devices with major and minor numbers `0,0`), created directories as regular directories, and overwritten or deleted directories as directories that have the `user.overlay.opaque` extended attribute set. To reuse these effects, INCR iterates over the upperdir and applies each change to the current environment.

Reusing prior results for non-deterministic or network commands may compromise correctness, as they may produce different outputs even with identical dependencies and input streams. While reusing prior results for non-deterministic commands may not necessarily sacrifice correctness (*e.g.*, for a command that fetches static web content), it is impossible to automatically determine which commands are always safe to incrementalize. Such behaviors fall outside INCR's scope; however, INCR accepts optional hints (§6.2) that allow developers to mark commands as non-incrementalizable.

5 Runtime Optimizations

INCR's tracing (§3) and memoization (§4) mechanisms may introduce runtime and storage overheads. This section describes several optimizations that make INCR practical for real-world workloads.

5.1 Eager Stream Processing

The shell's streaming execution model allows commands in a pipeline to start processing as soon as their upstream commands begin emitting output. This complicates incrementalization. Consider `awk '{print $1}' $f | grep 'x'`. If `$f` changes but its first column remains the same, then `awk`'s output is unchanged, meaning that `grep 'x'` can reuse its memoized results. However, awaiting each pipeline stage to decide reuse disables streaming semantics and incurs high overhead for long pipelines.

To address this challenge, INCR employs *eager stream processing*, a mechanism that skips redundant computation while preserving the shell's streaming model. When executing a pipeline, INCR's probes begin executing each command as soon as input is available and without waiting to check if re-execution can be skipped. Each probe buffers its `stdin` stream in memory while computing a rolling hash and forwarding it to the probe's command. Then, once the probe finishes hashing `stdin`, it checks if the command's inputs have changed. Importantly, probes typically finish hashing `stdin` *long before* their commands finish processing it, especially for compute-intensive commands, enabling them to determine if memoized results can be reused early in each re-execution run. If the command's dependencies and `stdin` have not changed, then INCR sends a `KILL` signal to the command and outputs its memoized results starting from where it left off. Otherwise, INCR continues running the command and tracking its effects as usual. This process continues in a chain for all the stages in a pipeline.

However, aborting a command that has filesystem side effects early may produce an inconsistent system state if it is in the middle of a modification. INCR's effect isolation mechanism (§3.2) addresses this: each command's effects are entirely contained within its OverlayFS upperdir. Therefore, INCR simply discards a command's upperdir when aborting its execution and applies its memoized effects instead.

5.2 Introspection

Effect isolation (§3.2) is necessary to contain and memoize side-effectful commands. However, creating, copying, and committing OverlayFS directories generates potentially expensive overheads. To reduce such overheads, INCR employs

introspection to detect, using knowledge from prior runs, whether commands need effect isolation. Specifically, INCR uses the tracing information to identify commands that do not perform filesystem modifications. INCR optimistically assumes that, given the same arguments, these commands remain effect-free in subsequent runs, similar to assumptions made by prior systems [48, 74]. Then, INCR skips effect isolation for these commands by running them without OverlayFS.

However, if such a command modifies the filesystem in a subsequent run—as detected by INCR's tracing mechanism—INCR *still* guarantees correctness. INCR completes the command's execution as normal, correctly applying its effects to the filesystem. It then revokes the command's effect-free designation and invalidates its cache entry to avoid reusing potentially stale effects in future runs. In the next run, INCR re-executes the command with effect isolation enabled.

Certain commands may be practically effect-free but still create temporary files during execution that are cleaned up before they exit. For example, `sort` creates temporary files if its input is too large to fit into memory. INCR assumes that files created and removed within the same command execution are temporary, and does not record them as dependencies. This approach allows INCR to consider such commands effect-free and to skip effect isolation in subsequent runs.

5.3 Storage Compaction

Memoizing dependencies and effects for commands that produce large outputs or for programs that consist of numerous commands can incur significant storage overheads. To mitigate this issue, INCR employs *storage compaction*, compressing memoized data with a configurable compression level. INCR uses the zstd compression algorithm, which offers extremely fast compression and decompression at reasonable compression ratios [23]. To avoid invalidating memoized data when the compression level changes, INCR records the compression level used for each memoized output. Therefore, INCR can decompress and reuse data generated under any previous configuration.

6 Optional Annotations and Configurations

INCR, as described so far, provides automated incremental execution for arbitrary shell programs. This section describes how INCR can leverage command annotations made available by other systems (§6.1) and optional developer configurations (§6.2) to achieve additional optimizations and to increase incrementalization fidelity.

6.1 Existing Crowdsourced Annotations

INCR leverages crowdsourced command annotations made available by other systems [43, 48, 66, 74, 86] to further increase the granularity of incrementalization. These annotations target parallelization and distribution opportunities, but the information they expose can benefit INCR's analysis and accelerated re-execution. Each annotation maps a command invocation to a set of properties. For example, a combined annotation from POSH [74] and PaSh [48] for `cat` is:

```
cat: [], stateless, splittable_args
```

This annotation indicates that `cat` is stateless and splittable across its arguments when invoked without any flags. The following are the properties that INCR can exploit.

INCR exploits *statelessness*, a classification from both POSH [74] and PaSh [86] for commands that operate on each input line independently without maintaining any internal state across lines. For example, invocations of `grep` without `-c` are classified as stateless over their `stdin` stream. This allows INCR to re-execute only the affected parts of a stateless command's input stream. It splits the inputs of stateless commands into smaller chunks and memoizes each chunk separately and in parallel. Data streams are split using content-defined chunking [89], which produces chunks that are stable across input perturbations. If a modification to a stateless command's input affects only a few chunks (*e.g.*, when a log file is extended with new events), then memoized results of unaffected chunks can be reused.

INCR exploits *purity*, a classification for commands that do not modify the filesystem outside of a defined set of inputs and outputs. This classification comes from PaSh's parallelizability annotations [86]. For example, an annotation for `grep -f p.txt` classifies it as pure with read dependencies from `stdin` and `p.txt`, and a write dependency to `stdout`. INCR skips effect isolation and tracing (§3.2) on pure commands. For many common commands such as `cat`, `grep`, and `tr`, this reduces INCR's cold-start overhead when introspection (§5) has not yet detected that the command is pure.

INCR exploits *argument independence*, a classification from POSH [74] for commands that can be executed independently for each argument. For example, an annotation for `grep 'p' f1 f2` classifies it as argument-independent across its arguments. Specifically, if `f2` changes, then only `grep 'p' f2` needs to be re-executed. INCR performs argument-level incrementalization on these commands by syntactically transforming each invocation: INCR splits the single large invocation into multiple invocations inside a subshell, each with a single argument. For example, it transforms the previous `grep` invocation into `(grep 'p' f1 ; grep 'p' f2)`. INCR then places separate probes on each invocation and memoizes and reuses their results independently.

6.2 Optional Developer Configuration

Developers can also optionally configure INCR for specific script fragments to further improve performance, exploiting knowledge of a script's behavior and use patterns. To support such configurations in a backward-compatible fashion, INCR

Table 1: **Benchmark summary.** Summary of all the benchmarks used to evaluate INCR and their characteristics. Benchmarks are categorized based on the delta type: addition (+), deletion (−), modification (~), or a combination thereof and the reason for the change: behavior (B), wrong command (C), wrong flag (F), exploration (E), summarization (S), optimization (O), LLM assistance (L), replacement (R), input update (I), debugging (D), aggregation (A), or visualization (V).

	Benchmark	Description	Input	# deltas	LoC	Deltas	Source
1	**dpt**	Segments and extracts images.	2.4 GB	9	84	O 2F I A F V 2C	[32,47]
2	**bio**	Extracts genome sequences.	3.5 GB	6	104	I B I D 2E	[18,46]
3	**dict**	Calculates top-n most frequent words.	30 MB	1	2	S	[11]
4	**ngram**	Extends indexing stage to 1–3 grams.	106 MB	2	58	2B	[49]
5	**uppercase**	Analyzes word distribution.	200 MB	1	18	B	[49]
6	**unixgame**	Solves a Unixgame question.	1.0 GB	5	6	2E E 2E	[71,84]
7	**nginx**	Detects broken links.	974 MB	21	243	7E 5A R 5D S F O	[74]
8	**weather**	Calculates temperature statistics.	887 MB	2	138	2E	[82]
9	**covid**	Calculates public transit statistics.	381 MB	4	48	E E E E	[83]
10	**spell**	Finds spelling errors.	3.1 GB	6	48	3D E 2B	[11]
11	**poet**	Extracts used and rhyming words.	1 GB	3	47	3E	[49]
12	**image**	Renames images with VLLM.	38 MB	5	45	2D 2D L O	[32,47]
13	**music**	Prepares music files for sharing.	16 MB	6	52	3L D O O	[74,75,81]
14	**beginner**	Inspects system logs.	974 MB	13	28	5F 4E A O E S	[28,74]

exposes a special annotation that it detects during parsing. These annotations instruct INCR either to disable incrementalization for a command or to group multiple commands together to memoize them as a single unit. Such annotations are expressed as assignments to the placeholder environment variable `INCR` before commands. For example, configuring INCR to skip `cat` and `grep` within a pipeline looks as follows:

```
INCR="s" cat f | INCR="s" grep . | tr a-z A-Z
```

Configuring INCR to group commands together by marking the first and last commands in the group looks as follows:

```
cat f | INCR="g" grep . | INCR="ug" tr a-z A-Z
```

Setting an unused environment variable before a command does not change its execution.

Disabling incrementalization reduces runtime overheads for program fragments that perform minimal computation or have complex side effects. For example, configuring INCR to skip `cat` and `grep` commands in the example above avoids effect memoization for trivial commands that emit large outputs, thereby eliminating both runtime and storage overheads. Furthermore, disabling incrementalization for user-provided commands that rely on randomness that INCR cannot detect (*e.g.*, pseudorandom generators) guarantees correctness.

Grouping commands reduces overheads for program fragments that are unlikely to benefit from fine-grained, individual incrementalization. Additionally, grouping a fragment that is considered final and not expected to change allows INCR to focus only on fragments that will benefit from incrementalization. For example, configuring INCR to group the pipeline above incurs effect isolation and tracing overheads only once for the entire sequence rather than once per command. This can significantly improve performance for pipelines composed of many inexpensive commands.

7 Evaluation

This section applies INCR to 14 incremental development scenarios (totalling 81 deltas) to characterize its re-execution benefits (§7.1), its runtime overheads (§7.2), its behavioral equivalence (§7.3), the effectiveness of its optimizations (§7.4), and the impact of optional annotations (§7.5).

Benchmarks: To evaluate INCR, we use the Koala benchmark suite [55] to identify real-world shell workloads, and collect modifications that reflect their development. The benchmarks span data processing, machine learning, and system administration, with input sizes ranging from a few to tens of gigabytes and consisting of a total of 81 deltas, summarized in Tab. 1. Incremental modifications include fixing wrong commands or arguments, exploring new functionality, and using LLMs to guide modifications. All changes arise from a diverse set of goals (summarized in Tab. 1), resulting in three types of deltas: in-place edits to existing commands (~), *e.g.*, changing command arguments; additions of new commands or stages (+); and removals of existing commands or stages from the script (−).

The **dpt** benchmark (discussed in §2) segments and classifies hieroglyph images from a single expedition. Its changes include optimizing by removing a redundant image-resizing stage (O), correcting `awk`'s printing format (2F), filtering out irregular files in the **for** loop condition (I), aggregating the classification results (A F), and finally visualizing classification results with two iterations (V 2C). Inputs total 2.4 GB, including model weights and ultra-resolution images.

The **bio** benchmark processes genomic data to extract chromosome-specific subsets using a six-stage pipeline. Changes include iterating over all BAM files instead of a hard-coded testing file (I), extracting per-chromosome reads (B), driving processing from an input file of population-sample pairs (I), ignoring malformed or incomplete entries in that list (D), and plotting summary statistics for per-sample coverage (2E). The script is applied on a corpus of genomic data comprising 15 samples and 3.5GB of aligned reads.

The **dict** benchmark counts the frequency of each word in a corpus using a four-stage pipeline. The change modifies the script to only output the top-n most frequent words, by adding a stage to `sort` words by frequency and a last `head` stage (S). The program operates over a large text corpus of 5.2M words, totalling 30MB of text from Project Gutenberg [61].

The **ngram** benchmark starts with a unigram computation pipeline of `tr`, `sort`, and `uniq`, extended to compute bigrams (B) and trigrams (B). It processes a 16-million-word, 106MB snapshot of Project Gutenberg [61].

The **uppercase** benchmark extracts all unique capitalized words from a large text corpus using multiple stages of `grep`, `tr`, and `sort`. The first delta modifies the script to count the occurrences of unique capitalized words by adding `sort -u` to sort the words by their frequency in descending order in the middle of the pipeline (B). Its input is a 33-million-word snapshot of Project Gutenberg, totalling 200MB [61].

The **unixgame** benchmark solves a series of questions from the Unix50th anniversary game. It first counts the total number of rounds using `grep '.'` and `wc -l`. Changes insert `grep 'x'` to capture moves from one side (E), then a `cut` and `grep -v '[KQRBN]'` to count specific captures (E), then `cut`, `sort`, and `uniq -c` to count occurrences of each capture (E), normalize lowercase identifiers via `tr '[a-z]' 'P'` (E), and counting occurrence frequency with `sort -r`, `uniq -c`, `head`, and `awk` (E). Inputs are a chess dataset from Lichess [26] totalling 19M chess moves and 1.0 GB.

The **nginx** benchmark analyzes Nginx server log entries to identify broken links. A series of changes expands its scope, *e.g.*, extracting status codes, listing request paths that lead to `402` or `502` errors, identifying suspicious requests, counting unique clients, extracting referrers, sorting and ranking error-inducing URLs, and summarizing top 404-error paths (7E 5A R 5D S). Another set modifies ordering and summarizing behavior to use consistent reverse-numeric sorting (F), and deleting unnecessary outputs (O). It operates on a 5-million-record, 974MB web-server log [74].

The **weather** benchmark processes a large weather dataset to compute maximum temperatures for each day between 1995 and 2000. The changes modify the script to add two additional statistics by computing the minimum (E) and average temperature (E). Its input is an 887MB weather dataset of 3.6M temperature records from the National Oceanic and Atmospheric Administration (NOAA) [2].

The **covid** benchmark analyzes public transit data collected during COVID-19 to compute a series of statistics. The changes introduce more metrics such as total vehicles per day, days per vehicle (E), hours per vehicle (E), monitored hours per day (E), and hours per bus (E). The last change introduces a single `awk` invocation that computes the statistics in a single pass. Its input contains 5M bus schedule records, totalling 381MB [83].

The **spell** benchmark analyzes spelling mistakes in a large text corpus. It includes six changes: removing non-printable characters and turning the input into a word stream (D), lowercasing that stream (D), removing punctuation (D), sorting words in alphabetical order (E), reporting words not found in a dictionary (B), finally comparing only unique words against a dictionary to identify misspelled words (B). The input is a collection of 9001 books, totalling 527M words and 3.1GB of text [61].

The **poet** benchmark counts the frequency of each word found in a given directory containing text files. Changes include replacing the single-file input with a concatenated corpus of all poetry files to enable global text statistics (E), adding a second output that reports unique words in alphabetical order (E), and finally introducing a third output that orders words by rhyme by reversing strings prior to sorting and then restoring their original orientation (E). The input is a corpus of 3001 books, totalling 22.4M lines and 1GB of text from Project Gutenberg [61].

The **image** benchmark renames images based on their content using a vision-language model—`GPT-4o mini`. Changes include replacing spaces in LLM-generated titles with underscores to form basic filenames (D), lower-casing all characters for consistency (D), stripping non-alphanumeric, non-underscore, and non-dash characters to guarantee filesystem-safe names (D), separating the cleaned stem into a reusable `base` variable for clearer filename construction (L), consolidating the sanitization steps into a single `sed` invocation (O), and finally removing the mode suffix so outputs use only the cleaned base title (D). Its input is a set of 11 images from a browsing session (totalling 38MB) [32].

The **music** benchmark captures a vibe-coding development loop, where a user iteratively refines a multimedia pipeline through loosely guided interactions with an LLM [36]. Starting from a simple `mp3` to `wav` conversion with `ffmpeg`, changes include `tar`-ing each newly produced `.wav` file alongside conversion so every MP3 immediately gets its own archive (L), refactoring into a single post-loop that aggregates all WAVs into a combined tarball (D), encrypting it with `openssl` (L), introducing a configurable encryption-key variable instead of a hardcoded one (O), compressing individual WAV files to `.gz` before archiving (L), and finally simplifying the layout by dropping per-file compression in favor of `gzip`-ing (O). Its input is a collection of 20 public-domain music files, totalling 16MB [55].

The **beginner** benchmark inspects system logs to identify failed login attempts. Starting from a numeric sort of the

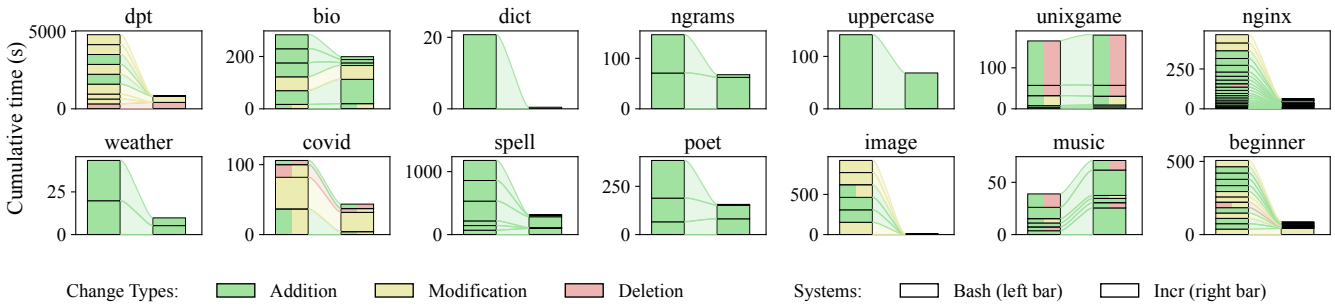

Fig. 4: **INCR's speedup on incremental changes**. Each vertical bar, *i.e.*, group of blocks, represents the execution time of a benchmark change during incremental development. Each block within a bar represents the time taken by the corresponding re-execution of the benchmark after a change. The curves connect the same block across the two systems.

system log, changes include switching from numeric sorting to lexicographic sorting (`F`), counting duplicate lines (`E`), filtering for lines containing specific patterns—first case-sensitive (`E`) and later case-insensitive (`F`), counting matching lines (`A`), then consolidating filtering and counting (`O`), extracting the first two fields from the counted output (`E`), extracting different fields (`F`), grouping identical field pairs after sorting (`E`), numerically sorting the grouped results (`E`), reversing the numeric sort to rank largest first (`A`), selecting the top ten entries (`S`), and adjusting the `head` call to the explicit `head -n 10` form (`F`). Its input contains 5 million system logs, totalling 974MB [74].

Experimental Setup: Experiments were conducted on a Cloudlab m510 machine with 8-core Intel Xeon D-1548 CPU at 2.0GHz, 64GB RAM, 256GB NVMe, and 10Gb connection, running Ubuntu 22.04 with Linux kernel 5.15.

7.1 Re-execution Performance

By how much does INCR accelerate re-execution?

Methodology: For each program modification, we measure the execution time of the script under INCR and Bash, and report the speedup as the ratio of Bash's runtime to INCR's runtime per incremental step. We run each re-execution 3 times and report the average.

Results: Fig. 4 presents INCR's speedup—applied fully automatically without annotations—over Bash across all benchmarks and incremental re-execution. Across all benchmarks and re-executions, INCR achieves an average speedup of 34.2×, with a maximum speedup of 373.3×. Out of 85 re-executions, INCR achieves speedup in 69 cases and slowdown in 16 cases. For speedup, INCR achieves an average speedup of 34.2× across all benchmarks, in best case 373.3×, and in worst case 1.003×. For slowdown, INCR incurs an average slowdown of 0.73× across all benchmarks, in best case 0.95×, and in worst case 0.15×.

Discussion: INCR's substantial re-execution speedups stem from its ability to track fine-grained dependencies and safely reuse previously computed results, thereby eliminating redundant computation. For example, INCR reuses LLM-generated image annotations in the **`image`** benchmark when incremental changes only modify the post-processing logic, reducing the execution time from 155.55 seconds to 1.62 seconds, achieving a speedup of 96.02×. INCR does not introduce a significant difference in accelerating different types of changes, primarily because INCR's fine-grained dependency tracking effectively identifies unaffected commands and reuses their results regardless of the type of modification.

INCR's overheads stem primarily from its use of system-call tracing and isolation, which together enable the system to capture fine-grained dependencies and manage memoized results. These fixed costs become more pronounced and produce minor slowdowns in benchmarks dominated by many short-lived commands or in those with complex dependency behaviors. For example, in the **`unixgame`** benchmark—which consists of a sequence of short-running commands such as `tr ' ' '\n'`—the final iteration modifies only the third command, requiring full re-execution of the remaining eight commands. For these commands, fixed costs make up a larger fraction of the processing time, resulting in an increase in execution time from 107.8 seconds to 123.6 seconds. Moreover, in the first iteration of the **`music`** benchmark, tracing the `ffmpeg` and `tar` commands increases execution time from 3.7s to 12.4s. These commands make many system calls and thus are expensive to trace.

7.2 Cold-Start Overheads

To achieve these speedups (§7.1), INCR uses tracing, isolation, and memoization mechanisms. By how much do these mechanisms slow down the initial execution of a script, *i.e.*, before it benefits from incremental re-execution?

Methodology: To characterize the cold-start overheads, we measure INCR's execution time relative to Bash across all benchmarks with INCR enabled but no modifications made to the script. This represents the worst-case scenario for INCR:

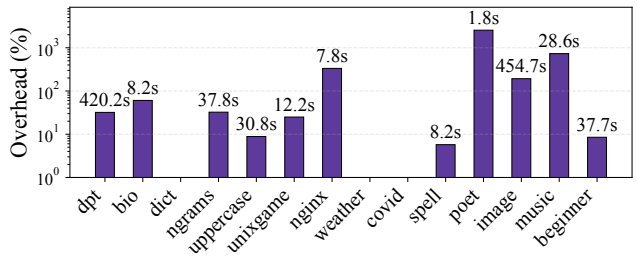

Fig. 5: **INCR's cold-start overhead**. Each bar represents INCR's overhead per benchmark during the first run. The x-axis represents benchmarks. INCR's absolute execution time is noted above each bar.

all tracing, isolation, and memoization overheads are incurred without any benefits from incremental execution.

Results: Fig. 5 presents INCR's overhead relative to Bash in the absence of any changes. For benchmarks where INCR's execution exceeds five seconds, INCR exhibits an average overhead of 101.05%, with overheads ranging from 732.45% at worst to -48.46% at best (a 1.92x speedup).

Discussion: INCR's overheads stem from its tracing, isolation, and memoization mechanisms. However, these overheads are often offset by the benefits of incremental execution in real-world development scenarios, where scripts are frequently modified and re-executed. INCR at times outperforms Bash even during the initial execution without incremental changes, primarily due to its runtime optimizations (§7.4), *e.g.*, eagerly executing commands to consume input streams.

7.3 Behavioral Equivalence

Does INCR preserve behavioral equivalence to an unmodified shell interpreter during incremental re-execution?

Methodology: This section characterizes INCR's behavioral equivalence to the underlying shell interpreter by applying it to the standard Bash test suite. Version `5.2.37(1)-release` consists of 83 test categories, each corresponding to a specific feature of Bash, totalling 534 test files and 22,064 LoC. The suite does not include named assertions; instead, it comes with ground-truth text files that include the expected outputs from each test invocation. Equivalence is defined as the system producing outputs identical to the ground-truth files when executing the tests. These files contain 10,282 lines, covering all Bash features ranging from variable expansion to job control. Matching the interpreter's behavior exactly is therefore unusually demanding: the oracle is sensitive to whitespace, quoting, error text, ordering, and many tests hinge on interpreter idiosyncrasies and historically accreted corner cases. As such, even semantically equivalent behavior can be flagged as incorrect unless it reproduces the ground-truth output verbatim. In fact, using vanilla Bash (v. 5.2.37) to run the test suite unveils 362 lines of output that differ from the ground-truth files. Instead, we use Bash as the ground-truth and compare INCR's outputs against it.

Table 2: **INCR's behavioral equivalence.** The table shows several Bash groups, example categories, and the number of tests per category as well as INCR's results.

Group	Example categories	INCR
Globbing	`extglob`, `globstar`	868/868 (100%)
Data structs.	`array`, `array2`, `assoc`	1261/1261 (100%)
Quoting	`quote`, `quotearray`	877/877 (100%)
Expansion	`comsub`, `alias`	1701/1703 (99.9%)
Utils	`getopts`, `strip`	80/80 (100%)
Env.	`attr`, `history`, `read`	1610/1610 (100%)
Constructs	`parser`, `func`, `case`	1547/1547 (100%)
IPC	`jobs`, `execscript`	1355/1356 (99.9%)
POSIX	`posixexp`, `posixpipe`	536/536 (100%)
Options	`set-e`, `set-x`, `shopt`	444/444 (100%)
Total		10,279/10,282 (99.9%)

A single Bash test file includes multiple test cases. Examples of such cases include how temporary environment assignments affect only a child process's environment, while expansions use the caller's variable values (*e.g.*, `VAR=abc /bin/echo $VAR` prints nothing), that `$!` expands to the PID of the last process substitution (*e.g.*, `cmd <(exit 123)`) and that using `wait` returns that subprocess's exit code. The tests also invoke external utilities like `grep`, `cat`, and `awk`.

A set of 19 cases are considered out of INCR's scope: while INCR correctly parses and reports such errors before execution, its parsing diverges subtly due to its underlying `libbash` parsing library. Some error messages differ in the reported line number, due to `libbash`'s lossy pretty-printing of the Bash AST; other messages differ in the name of the interpreter, and non-UTF-8 escape sequences in strings are misprinted due to lacking `libbash` support.

Aside from the standard Bash tests, we also inspect the execution and results of all real-world scenarios (Tab. 1) with INCR. Using the same experimental setup as in §7.1, we confirm that (1) the final outputs produced by both INCR and Bash match those by the authors of the Koala suite (*i.e.*, files read by the benchmark's `verify.sh` script); and (2) all exit codes executed from INCR and Bash after each incremental modification match each other.

Results: Applied to the Bash test suite, INCR only differs by 3 ground-truth lines out of 10,282 (99.9% equivalence, Tab. 2). Differences come from (1; 2 diffs) recursive `alias` definitions, which INCR cannot identify during probe placement, thus placing a probe to the aliased command whose expansion (unlike normal shell expansion) INCR cannot observe; (2; 1 diff) an `execscript` test that unsets `PATH`, causing INCR to fail to locate its own dependencies. We re-ran all tests without clearing INCR's cache, obtaining identical results, to confirm reuse does not affect equivalence.

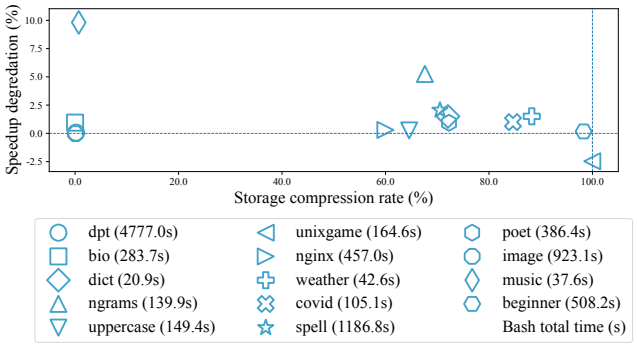

Fig. 6: **INCR's storage compaction impact on performance and storage**. Each point represents the storage reduction *vs.* speedup degradation per benchmark, compaction enabled.

Discussion: There are several successful cases worth highlighting. INCR had to be extended to support all the different ways Bash can be invoked—including argument transfer and correct escaping, which had to go through the shell interpreter, INCR itself, and the underlying effect isolation INCR uses. Additional support had to do with identifying, before probe placement, all the built-in Bash constructs that do not correspond to trackable external binaries in the system. INCR also had to be extended to track a variety of additional effects such as permission modification, movement, and creation of symbolic filesystem links.

INCR exhibits two classes of divergences, namely in `alias` and `execscript`, which arise when scripts manipulate the shell environment in ways that prevent INCR from resolving script contents or locating its own runtime dependencies. Several approaches could be used to address these limitations, including deferring probe placement and decoupling dependency resolution from the runtime environment.

7.4 Effects of Runtime Optimizations

What are the benefits of various optimizations, including eager stream processing, introspection, storage compaction, stream size heuristics, and optional annotations and configurations?

Eager stream processing: This optimization is always enabled (§7.1); when disabled, INCR waits for each command to finish before deciding whether the subsequent command needs to be executed.

We apply INCR to a synthetic program that executes a single 16-stage pipeline twice without any incremental change; all stages are streaming commands, *e.g.*, `grep` filtering lines and `sed` performing text substitutions. On the script's first execution, INCR takes 3m 22s with eager stream processing, reducing the execution time by 65.8% from 9m 50s without eager stream processing. On the script's second execution, INCR takes 10s to fetch the memoized results both with and without this optimization. Eager stream processing adds imperceptible overhead on reuse.

Introspection: This optimization is always enabled (§7.1); when disabled, INCR always re-executes commands within an isolation sandbox during incremental runs whether or not they have write dependencies.

We apply INCR to a synthetic script that includes a pipeline consisting of 20 commands that communicate only through their standard streams. Each incremental change modifies the input data processed by the pipeline, triggering re-execution of all commands. Without introspection, INCR takes 35s on all iterations. With introspection, INCR still takes 35s on the first iteration because it has not yet detected that the commands are pure, but only 31s (speedup: 13%) on subsequent iterations.

Storage compaction: This optimization is disabled by default (§7.1); when enabled, INCR compresses memoized dependencies and effects on disk, achieving significant space savings with slightly increased runtime overheads. Fig. 6 shows that compaction reduces space usage across all benchmarks by an average of 55.7%, with a maximum of 100.0%, while trading off a 1.9% average speedup degradation and up to 9.8% in runtime performance. This optimization is especially beneficial for scripts that include longer pipelines, as each intermediate result is a separate cache entry.

7.5 Effects of Optional Annotations

How do optional crowdsourced annotations (§6.1) help INCR further accelerate incrementalization?

Methodology: We apply INCR with annotations to all benchmarks in Tab. 1. Additionally, we design two synthetic shell programs. The first program uses a five-stage pipeline of stateless commands. Changes append new lines to the input file. The second program invokes two argument-independent commands—`clang` and `sha256sum`—to compile and hash C files. Changes modify one of these C files.

Results: With annotations, INCR introduces an additional average speedup of 1.46× across all benchmarks, and up to 24.40× in the **`music`** benchmark. INCR lowers the cold-start overheads from an average of 101.05% to 43.55% and from a maximum of 732.45% to 278.15%. In the first synthetic program, INCR takes 29s on the first run (2.51×), and only 6s on the next two runs (12.17×), compared to 1m 13s on each run without annotations. In the second synthetic program, INCR takes 22.0s (4.9×) after changes, compared to 1m 48.8s without annotations.

Discussion: Annotations allow INCR to make informed decisions based on command semantics. For example, in the **`music`** benchmark, annotations indicating that the `ffmpeg` and `tar` commands are pure allow INCR to skip expensive tracing of these commands. Moreover, INCR applies chunked incrementalization for stateless commands in the first synthetic program, and decomposes each command's argument list into per-file invocations in the second synthetic program.

8 Related Work

Incremental computation: Incrementalization systems [5, 41, 42, 57] typically employ language-level approaches by tracking fine-grained dependencies between program fragments, requiring language support and modifications to the program source code to expose dependency boundaries. INCR instead discovers dependencies at the *system level* through effect tracking—bringing incremental behavior to environments that span multiple languages and include opaque components.

Data processing systems [21, 65, 72, 91] support incremental computation through dataflow models or adjacent domain-specific computation models. In contrast, INCR targets general-purpose shell environments that span opaque components and arbitrary data-processing semantics, and instead focuses on automatic incrementalization without relying on specialized dataflow abstractions or domain-specific processing models.

Build systems: Build systems [33, 53, 59, 62] aim to efficiently rebuild software after source code changes, often requiring developers to manually specify their dependency graphs. Prior research reduces the burden of manual dependency specification in build systems by automatically inferring dependencies through execution tracing, thereby enabling incremental builds without explicit declarations [10,14,25,78]. INCR fundamentally differs from these systems by targeting general computations that span data dependencies, transient effects, and mutable state that go beyond build artifacts.

Riker [25] infers fine-grained dependencies from simple build scripts (Rikerfiles) written in any language and delivers precise, language-agnostic incremental builds. It focuses on accelerating re-execution when inputs change. INCR captures changes arising not only from inputs, but from arbitrary shell program modifications, including both changes on persistent artifacts (*e.g.*, files and directories), transient effects (*e.g.*, data streams), and mutable state (*e.g.*, environment variables).

Exploratory programming: Interactive environments such as notebooks [51] shorten the development feedback loop by exposing cell-based re-execution, but require manual control of dependencies [92]. For example, systems such as Jupyter [1, 3, 54] allow users to (re-)execute program cells independently, but require them to manually manage dependencies across coarse-grained cells to ensure correctness during re-execution. INCR instead infers dependencies automatically.

Reactivity and view maintenance: Reactivity and view maintenance have been studied in both theory [15, 40, 56] and practice [6, 31, 90, 91]. Reactivity's goal is to automatically identify and execute the minimal re-computation needed to maintain up-to-date results as input data changes. Prior systems have focused on reactivity within specific domains, such as relational databases [19], or map-reduce-style data processing [27]. INCR does not target reactivity.

Provenance tracking and reproducibility: Provenance systems [9, 20, 64, 70, 77] capture dependencies among entities to support auditing and forensics. INCR's goals and methods for extracting dependencies in shell programs fundamentally differ, but the provenance graphs produced by these systems are complementary to INCR's dependency tracking and could enable incremental execution across network boundaries.

Prior research explores re-execution across diverse environments by caching execution dependencies and intermediate results. For example, package-management systems such as CDE [39] and ReproZip [22] interpose on system calls to record the resources accessed during a program's execution, enabling the program to be re-executed on a different machine by packaging all recorded dependencies. These systems primarily target portability of executions across environments, rather than incremental execution, as INCR does.

Systems such as Nix [29], Guix [24], and Docker [60] enable users to declaratively specify system and application dependencies to improve reproducibility and deployment reliability. Although these systems support incremental builds and deployments, they rely on explicit dependency specifications to accelerate re-execution, while INCR does so automatically.

Other systems [12, 44, 67] record program execution to enable faithful replay on often unmodified applications. These systems focus primarily on deterministic re-execution for debugging and reproducibility, whereas INCR targets accelerated incremental execution in environments that treat arbitrary commands as the primary unit of computation.

Research on the Shell: Recent systems such as POSH [74], PaSh [48], and Fractal [43] automatically parallelize or distribute shell programs. These systems accelerate large-scale computations and rely on developer annotations to identify parallelizable and distributable fragments. In contrast, INCR targets early-stage exploratory development and reduces re-execution time by automatically inferring dependencies and memoizing command effects at runtime. It can optionally leverage insights from these systems to increase incrementalization fidelity (§6.1).

Furthermore, a variety of systems improve the shell along many dimensions, including syscall refinement [34], fusion [8], elision [13], dataflow extension [81], synthesis [76], serverless execution [58], and mobile usage [88]. INCR is complementary to these works.

9 Conclusion

Fundamentally, INCR shows that bolt-on incrementalization atop unmodified shells is not only possible, but also broadly applicable. Its design includes lightweight effect tracking, safe memoization and reuse, optimizations that address key performance bottlenecks, and configurations that enable further incrementalization opportunities. Applied to real-world workflows, INCR delivers substantial re-execution speedups without program or environment modifications and retains behavioral equivalence to normal execution.

References

[1] marimo. https://marimo.io/. Accessed: 2025-12-01.

[2] National Oceanic and Atmospheric Administration (NOAA). https://www.noaa.gov. Accessed: 2025-01-13.

[3] Vizierdb. https://vizierdb.info/. Accessed: 2025-12-01.

[4] Digital Pyramids Text Project, 2024. Accessed: 2025-10-20.

[5] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 14–25, New York, NY, USA, 2003. Association for Computing Machinery.

[6] Yanif Ahmad and Christoph Koch. Dbtoaster: a sql compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.*, 2(2):1566–1569, August 2009.

[7] Christelle Alvarez. The variability of ritual texts: Knowledge transfer at the interface of oral and written forms. In *Variability in the Earlier Egyptian Mortuary Texts*, pages 219–249. BRILL, October 2023.

[8] Anna Herlihy and Periklis Chrysogelos and Anastasia Ailamaki. Boosting Efficiency of External Pipelines by Blurring Application Boundaries. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[9] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 319–334, USA, 2015. USENIX Association.

[10] Ben Hoyt and Simon Alford. Fabricate, 2020.

[11] Bentley, Jon and Knuth, Don and McIlroy, Doug. Programming pearls: a literate program. *CACM*, 29(6):471–483, June 1986.

[12] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 177–191, USA, 2010. USENIX Association.

[13] Emery D. Berger. Optimizing Shell Scripting Languages. Technical Report UMCS TR-2003-009, University of Massachusetts Amherst, 2003.

[14] Bill McCloskey. Memoize: A replacement for make, 2008. Archived: 2010-09-05.

[15] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[16] Neil Brown, Miklos Szeredi, Amir Goldstein, Vivek Goyal, Randy Dunlap, Linus Torvalds, Pavel Tikhomirov, Kevin Locke, Sargun Dhillon, Chengguang Xu, and Deming Wang. The overlay filesystem. *The Linux Kernel documentation*, 2022. Started in 2014.

[17] French-Owen Calvin. Reflections on Building with OpenAI's API, 2024. Accessed: 2025-10-23.

[18] Cappellini, Enrico and Welker, Frido and Pandolfi, *et al.* Early Pleistocene enamel proteome from Dmanisi resolves Stephanorhinus phylogeny. *Nature*, 574(7776):103–107, Oct 2019.

[19] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery.

[20] Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. Data provenance at internet scale: Architecture, experiences, and the road ahead. In *8th Conference on Innovative Data Systems Research (CIDR '17)*, January 2017.

[21] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 85–98, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. Reprozip: Computational reproducibility with ease. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2085–2088, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Yann Collet. Zstandard: Fast real-time compression algorithm. https://github.com/facebook/zstd, 2016. Accessed: April 8, 2026.

[24] Ludovic Courtès. Code staging in gnu guix. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, page 41–48, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Charlie Curtsinger and Daniel W. Barowy. Riker: Always-Correct and fast incremental builds from simple specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 885–898, Carlsbad, CA, July 2022. USENIX Association.

[26] Datasnaek. Chess Games Dataset, 2020. Accessed: 2025-10-15.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[28] Diomidis Spinellis. DelftX: Unix Tools: Data, Software and Production Engineering. https://www.edx.org/learn/unix/delft-university-of-technology-unix-tools-data-software-and-production-engineering. Accessed: 2025-01-13.

[29] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, page 79–92, USA, 2004. USENIX Association.

[30] Mokhtar Ebrahim and Andrew Mallett. *Mastering Linux shell scripting*. Packt Publishing, Birmingham, England, 2 edition, April 2023.

[31] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997.

[32] Evangelos Lamprou. Foundation Models and Unix, March 2025.

[33] Free Software Foundation. Gnu make. https://www.gnu.org/software/make/make.html, 2023. Accessed: 2025-10-22.

[34] Gaidis, Alexander J. and Atlidakis, Vaggelis and Kemerlis, Vasileios P. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1964–1978, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Jie Gao, Simret Araya Gebreegziabher, Kenny Tsu Wei Choo, Toby Jia-Jun Li, Simon Tangi Perrault, and Thomas W Malone. A taxonomy for human-llm interaction modes: An initial exploration. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '24, New York, NY, USA, 2024. Association for Computing Machinery.

[36] Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, Baolong Bi, Fangda Guo, Jiafeng Guo, Shenghua Liu, and Xueqi Cheng. A survey of vibe coding with large language models, 2025.

[37] Aurelien Geron. *Hands-on machine learning with scikit-learn, keras, and TensorFlow*. O'Reilly Media, Sebastopol, CA, 2 edition, October 2019.

[38] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix shell programming: The next 50 years. In *Workshop on Hot Topics in Operating Systems*, HotOS '21, page 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Philip J. Guo and Dawson Engler. Cde: using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, page 21, USA, 2011. USENIX Association.

[40] Ashish Gupta and Inderpal Singh Mumick. *Maintenance of materialized views: problems, techniques, and applications*, page 145–157. MIT Press, Cambridge, MA, USA, 1999.

[41] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 156–166, New York, NY, USA, 2014. Association for Computing Machinery.

[42] Roger Hoover. Alphonse: incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 261–272, New York, NY, USA, 1992. Association for Computing Machinery.

[43] Zhicheng Huang, Ramiz Dundar, Yizheng Xie, Konstantinos Kallas, and Nikos Vasilakis. Fractal: Fault-tolerant shell-script distribution. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, Renton, WA, May 2026. USENIX Association.

[44] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. Ddos: taming nondeterminism in distributed systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 499–508, New York, NY, USA, 2013. Association for Computing Machinery.

[45] Jeroen Janssens. *Data science at the command line*. O'Reilly Media, Sebastopol, CA, October 2014.

[46] Jon Puritz. Bio594: Using genomic techniques to examine the evolution of populations. https://git.io/JY6J7, 2019.

[47] Justine Tunney. Bash One-Liners for LLMs. https://justine.lol/oneliners, 2023. Accessed: 2025-06-01.

[48] 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 769–785, Carlsbad, CA, July 2022. USENIX Association.

[49] Kenneth Ward Church. Unix for Poets, 1994.

[50] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1984.

[51] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything, 2023.

[53] Kitware. CMake. https://cmake.org/. Accessed: 2025-12-05.

[54] Thomas Kluyver, Benjain Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter Notebooks—a publishing format for reproducible computational workflows. pages 87–90. IOS Press, 2016.

[55] 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.

[56] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[57] Yanhong A. Liu. Incremental computation: What is the essence? (invited contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, PEPM 2024, page 39–52, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Mahéo, Aurèle and Sutra, Pierre and Tarrant, Tristan. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, Middleware '21, page 9–15, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Evan Martineau. Ninja: A small build system with a focus on speed. https://ninja-build.org/, 2012. Accessed: 2025-02-14.

[60] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[61] Michael S. Hart and Project Gutenberg. Project Gutenberg. https://www.gutenberg.org, 1971.

[62] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. Non-recursive make considered harmful: build systems at scale. *SIGPLAN Not.*, 51(12):170–181, September 2016.

[63] Jason Morris, Chris McCubbin, and Raymond Page. *Hands-On Data Science with the Command Line*. Packt Publishing, Birmingham, England, January 2019.

[64] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, page 4, USA, 2006. USENIX Association.

[65] 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, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[66] 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.

[67] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. Reproducible containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 167–182, New York, NY, USA, 2020. Association for Computing Machinery.

[68] Evi Nemeth, Garth Snyder, Trent R Hein, Ben Whaley, and Dan Mackin. *UNIX and Linux System Administration Handbook*. Addison-Wesley Educational, Boston, MA, 5 edition, August 2017.

[69] Addy Osmani. *Beyond Vibe Coding: From Coder to AI-Era Developer*. O'Reilly Media, 2025.

[70] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 405–418, New York, NY, USA, 2017. Association for Computing Machinery.

[71] Pawan Bhandari. Solutions to unixgame.io. https://git.io/Jf2dn, 2020. Accessed: 2020-04-14.

[72] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 251–264, USA, 2010. USENIX Association.

[73] David Quigley, Josef Sipek, Charles P Wright, and Erez Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006.

[74] Raghavan, Deepti and Fouladi, Sadjad and Levis, Philip and Zaharia, Matei. POSH: a data-aware shell. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[75] Schröder, Michael and Cito, Jürgen. An empirical investigation of command-line customization. *Empirical Software Engineering*, 27(2):30, 2021.

[76] Shen, Jiasi and Rinard, Martin and Vasilakis, Nikos. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 431–432, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[78] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[79] Diomidis Spinellis. Outwit:{UNIX}{Tool-Based} programming meets the windows world. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*, 2000.

[80] Diomidis Spinellis and Georgios Gousios. How to analyze git repositories with command line tools: we're not in kansas anymore. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 540–541, New York, NY, USA, 2018. Association for Computing Machinery.

[81] Spinellis, Diomidis and Fragkoulis, Marios. Extending Unix Pipelines to DAGs. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[82] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc, 2009.

[83] Tsaliki, Eleftheria and Spinellis, Diomedes. The Real Numbers for Athens Buses, 2020.

[84] Unix Game. The Unix Game - 50 Challenges to Master the Command Line, 2024. Accessed: 2025-10-19.

[85] Jake VanderPlas. *Python Data Science Handbook*. O'Reilly Media, Sebastopol, CA, December 2016.

[86] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *16th European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[87] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Towards fine-grained, automated application compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS '17, page 43–50, New York, NY, USA, 2017. Association for Computing Machinery.

[88] Winstein, Keith and Balakrishnan, Hari. Mosh: an interactive remote shell for mobile clients. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 15, USA, 2012. USENIX Association.

[89] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient Content-Defined chunking approach for data deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 101–114, Denver, CO, USA, June 2016. USENIX Association.

[90] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[91] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery.

[92] Megan Zheng, Will Crichton, Akshay Narayan, Deepti Raghavan, and Nikos Vasilakis. When are reactive notebooks not reactive?, 2025.