A Benchmark Suite for Research Targeting the POSIX Shell: Characterization and Implications

EVANGELOS LAMPROU, Brown University, USA
ETHAN WILLIAMS, Brown University, USA
GEORGIOS KAOUKIS, Archimedes, Athena Research Center, Greece
ZHUOXUAN ZHANG, Brown University, USA
MICHAEL GREENBERG, Stevens Institute of Technology, USA
KONSTANTINOS KALLAS, University of California, Los Angeles, USA
LUKAS LAZAREK, Brown University, USA
NIKOS VASILAKIS, Brown University, USA

Koala is a benchmark suite for research targeting the Unix and Linux shell, an environment for composing heterogeneous, opaque components. Koala combines a systematic collection of diverse shell programs collected from tasks found in the wild, various real inputs to these programs that facilitate small and large deployments, extensive analysis and characterization aiding their understanding, and additional infrastructure and tooling aimed at usability and reproducibility in systems research. The Koala benchmarks perform a variety of common shell tasks; they combine all major language features of the POSIX shell; they use a variety of POSIX, GNU Coreutils, and third-party components; and they operate on inputs of varying size and composition—available on both permanent archival storage and scalable cloud storage. Applying Koala to six systems that affect shell program execution, ranging from the underlying interpreter to bolt-on incremental execution, offers a broader perspective on their trade-offs, generalizes their key results, and contributes to a better understanding of these systems.

Additional Key Words and Phrases: benchmarks, performance, shell scripting, Unix/Linux shell

1 Introduction

Shell programming is as prevalent as ever. It consistently ranks among the top ten programming languages, with a recent popularity increase that dwarfs those of established languages such as C and Python [29]. These trends are mirrored by recent academic activity on the shell aimed at accelerating shell programs through parallelization [46, 83, 95], distribution [60, 79], and other forms of scale-out [31, 52, 57, 84].

Unfortunately, research on the shell is often held back by the absence of an established benchmark suite—a systematic collection of representative, diverse, and well-studied programs released as a reusable artifact. Consider the difficulties that the Shark authors faced in evaluating the benefits of its program transformations [4]:

> *To our knowledge, there does not exist a set of shell language benchmarks. We present preliminary results on a handful of microbenchmarks that we wrote ourselves.*

The absence of such a suite decelerates research on the shell, as authors waste time searching for new benchmarks; eschews core scientific principles, such as replicability and reproducibility, and hinders fair comparison, as different systems are compared against different baselines; creates

Authors' Contact Information: Evangelos Lamprou, Brown University, Providence, RI, USA, vagos@lamprou.xyz; Ethan Williams, Brown University, Providence, RI, USA, ethan@ethan.ws; Georgios Kaoukis, Archimedes, Athena Research Center, Athens, Greece, gkaoukis@cslab.ece.ntua.gr; Zhuoxuan Zhang, Brown University, Providence, RI, USA, zhuoxuan_zhang@brown.edu; Michael Greenberg, Stevens Institute of Technology, Hoboken, NJ, USA, michael@greenberg.science; Konstantinos Kallas, University of California, Los Angeles, Los Angeles, CA, USA, kkallas@ucla.edu; Lukas Lazarek, Brown University, Providence, RI, USA, lukas_lazarek@brown.edu; Nikos Vasilakis, Brown University, Providence, RI, USA, nikos@vasilak.is.

additional and unnecessary work, as it forces researchers to hand-roll their own benchmarks; and limits the impact of otherwise sound and applicable techniques, given lack of supporting evidence.

The Koala benchmarks: This paper presents Koala, a benchmark suite aimed at research targeting environments that compose heterogeneous, opaque components, orchestrated by the Unix or Linux shell. The term *opaque* refers to components whose internal behavior is not available to, and cannot be directly reasoned about by, the runtime itself (*e.g.*, external commands). Koala combines a collection of diverse, real-world, POSIX shell programs; realistic inputs of varying size; extensive analysis and characterization of these benchmarks; and additional infrastructure for packaging, automation, and reporting aimed at reusability and reproducibility. Its goal is to enable and support research on general optimizations on shell programs, as well as broader systems research within the context of the shell and, more generally, computations that cross component boundaries—*e.g.*, analytics, automation, exploratory computing, network monitoring, continuous integration and delivery *etc.* (see §2–3).

The Koala benchmarks are sourced from multiple time periods and perform a wide variety of tasks typically found in the shell—including log analysis, data processing, system administration, bioinformatics, software building, and continuous integration and deployment. They combine all major language features of the POSIX shell; use a variety of POSIX, GNU Coreutils, and third-party commands; and operate on inputs of varying size and composition—available via both permanent archival storage and scalable cloud storage for rapid access. Additional automation aims at setting up Koala across a variety of environments, confirming input and output correctness, and reporting on several execution characteristics. This additional automation is structured to maximize usability, configurability, replicability, and reproducibility.

Extensive analysis of the Koala suite reveals that it covers the full range of features of the POSIX shell; represents these features in proportions that align with real-world shell scripts; captures a wide range of script sizes, complexities, and styles—reflecting the shell's diverse use cases; and exhibits a variety of runtime characteristics—from compute- to memory- to I/O-intensive and short-lived to long-running computations.

In summary, Koala makes the following contributions. First, a systematic and diverse collection of real-world programs representative of tasks commonly found in the context of the shell and combining a variety of computational domains, language features, and shell components. Second, a set of real-world inputs stressing these benchmarks as well as smaller inputs aimed at immediate results, plus curated change sequences (edits and input updates) that model incremental development, all stored on highly available and scalable cloud storage backed up by permanently available archival storage. Third, additional infrastructure offering tooling, automation, and configurability for executing these benchmarks on a variety of environments, confirming input and output correctness, and reporting on their observed characteristics; and Finally, extensive analysis and characterization of these benchmark programs and their inputs over several dimensions, and application of the entire suite on six prior systems and tools targeting the shell.

Availability: The full set of benchmarks, their inputs, and infrastructure for automation, containerization, and reporting are all available as an MIT-licensed open-source repository.

https://kben.sh

2 Example & Motivation

Koala targets the systematic evaluation of shell-related systems and tools—for example: `zsh` [24], an alternative shell interpreter; Shark [4], a system accelerating shell script execution using syntactic transformations; GNU `parallel` [86], a command-line utility parallelizing shell pipelines; PaSh [46],

```sh
#!/bin/sh
d=~/imgs; t=$(mktemp)
for img in $(ls $d/*.jpg); do
  title=$(llm "A title for this:" -a "$img")
  printf "%s\n" "$img" >> $t.i
  printf "%s\n" "$title" >> $t.t
done

cat $t.t | tr '[:upper:]' '[:lower:]' |
  sed 's/ /_/g' |
  sed 's/[^a-z0-9_-]//g' > $t.t.f

paste -d $'\t' $t.t.f $t.i |
  while IFS=$'\t' read -r title img; do
      mv "$img" "$d/${title}.jpg"
  done
```

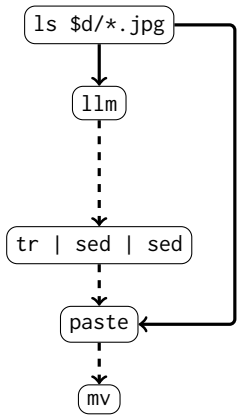

Fig. 1. Image renaming script. The script uses a vision-language model to rename images in a directory based on their content. The right graph illustrates the dataflow between components inside the shell program; dashed lines indicate streaming computation, while solid lines indicate batch computation.

a just-in-time shell-script parallelization system, *hS* [52], a system speculatively re-ordering shell program execution; and Incr [102], an incrementalization layer for shell programs.

Available options and the pains of characterization: The authors of these systems currently have a limited set of options for characterizing their benefits.

Microbenchmarks—small, isolated code fragments—are necessary for zooming into key trade-offs, stressing particular behaviors, and highlighting system limits—and are often handwritten to meet these goals [4, 46, 60, 61, 95]. Such synthetic workloads differ significantly from ones seen in practice and don't admit generalizable conclusions or holistic evaluations of complex systems.

Standards tests [30, 35] offer a thorough evaluation of shell behavior—*e.g.*, that pipelines take precedence over redirections in their constituent commands. The POSIX test suite is one example [35]. They focus on behavioral equivalence against a specification, and do not represent real workloads. Lacking the size and features of workloads seen in practice, they offer little insight into how optimization systems affect real programs.

Open-source code repositories are ripe targets for longitudinal studies—*e.g.*, understanding a community's use of certain shell features [20, 81]. Leaving scraping and parsing challenges aside, these programs typically lack inputs, setup scripts, explicit dependency declaration, and portable constructs—often consisting of noisy, low-quality, or even incomplete programs. Ad hoc collections are also not well understood by the community or the authors themselves, who risk accidentally skewing results due to these challenges.

User or developer study corpora are primarily focused on understanding developer patterns [13, 33]. Due to time constraints and the need to control for confounding factors, user study corpora broadly optimize for program characteristics that do not necessarily support generalization to the diversity of size, shape, and complexity of real-world programs.

Requirements and goals: The aforementioned options available to the authors of systems such as `zsh`, Shark, `parallel`, PaSh, *hS*, and Incr are ill-suited for characterization.

For a set of benchmarks to support research on the shell and related systems, it needs to meet the following criteria: (1) real-world programs performing real computations on real inputs; (2) diversity in workloads, domains, shell features, and commands used; (3) automated setup, analysis, validation, and reporting; (4) community-wide understanding through extensive analysis and characterization;

(5) permanent and scalable availability, allowing anyone to download and use them. Koala meets all of these criteria.

Using Koala: The authors of these systems enjoy a variety of options for downloading and setting up Koala:

```
$ curl -s up.kben.sh | sh && ./koala/main.sh --min --bare
```

This option sets up platform-specific dependencies, then downloads or generates a minimal set of inputs (§3). It then executes the full suite using its default parameters, such as the default shell interpreter—except for (1) `--min`, which uses minimal inputs (just enough to exercise the entire suite, all output validators, and its reporting infrastructure), and (2) `--bare`, which executes in the current environment instead of a Docker container launched afresh for the execution of each benchmark. Koala then confirms output correctness, guarding against failures and cross-run interference, and generates a report of the entire execution. On a fresh AWS `t3.large` instance, such a minimal bare-metal setup, execution, and reporting completes in about 20 minutes.

Omitting the `--bare` launches Koala in a Docker container, avoiding cross-run contamination and permanent effects on the host environment beyond result creation (*e.g.*, dependency installation), and omitting `--min` executes Koala on normal-sized (small) inputs, revealing how a system under evaluation is affected by realistic workloads. Koala aggregates the results over multiple runs (five by default), increasing statistical confidence, and reports the results of the execution. Using the default parameters in the same environment and full inputs bumps execution to about 24 hours.

An example benchmark: To understand how systems such as `zsh`, Shark, `parallel`, PaSh, *hS*, and Incr would benefit from Koala, consider zooming into a single benchmark. Fig. 1 provides a real-world program (simplified for this discussion) iterating over a large set of images, renaming them based on their content using a vision-language model (*c.f.*, Tab. 1). The first part of the script iterates over all images in a directory, querying a large language model (LLM) for a title for each image, and storing the results in two temporary files; the second part processes the titles to create filesystem-safe filenames; and the third part pastes the two files together and renames each image accordingly. The `parallel` and Shark authors would need to apply manual rewrites for their systems; the rest of the authors could point `KOALA_SHELL` to their system (§4). They would then execute this benchmark on `small` (but, contrary to `--min`, *real*) inputs:

```
$ ./main.sh --small inference
```

Each of these systems accelerates **`inference`** differently. The `zsh` interpreter executes the script as is, affecting performance of shell built-in operations like variable assignment and the **`for`** loop. Shark executes iterations of the **`for`** loop in parallel, and eliminates the `cat` at the start of the second pipeline by moving inputs to `tr`. GNU `parallel` can be applied to both the first loop and the two pipelines; the user would need to identify which parts of the script to parallelize. PaSh parallelizes each pipeline stage. *hS* runs commands speculatively, unrolling loop iterations to execute in parallel. Incr caches intermediate results of each loop iteration and pipeline stage to avoid re-computation on repeated runs. The six systems exploit different opportunities, optimizing different shell constructs and leaving different parts of each script unchanged.

On commodity infrastructure (§7), speedups average 0.98× for `zsh`, 2.89× for Shark, 2.71× for `parallel`, 1.08× for PaSh, 1.78× for *hS*, and 19.15× for Incr. In contrast to similar numbers potentially collected over microbenchmarks, the composition and complexity of these benchmarks demonstrate the anticipated benefits—as well as potential limitations—of these systems in real-world settings.

Tab. 1. Koala benchmark summary. The table summarizes all benchmarks in the Koala suite. Col. 1: Identification characteristics, listing each benchmark set's name and (a subset of) the programs it contains. Cols. 2–4: Descriptions, summarizing application domains ($\mathcal{D}$), the number of programs in the set (#.sh), and the total lines of code (LoC). Col. 5: Inputs, summarizing the size of each benchmark's inputs. Cols. 6–7: Static features, summarizing syntactic characteristics—*i.e.*, the number of shell constructs (#Cons) and the number of distinct commands (#Cmd). Cols. 8–11: Dynamic features, summarizing the execution characteristics—*i.e.*, time spent on shell evaluation (t_S), time spent on commands (t_C), memory consumption (Mem), and total input-output (I/O). Cols. 12–13: System, summarizing the number of system calls (#SC) and open file descriptors (#FD). Col. 14: Source, containing a reference to the source of the benchmark.

Benchmark/Script	Surface			Inputs		Syntax		Dynamic				System		Source
	$\mathcal{D}$	#.sh	LoC	Small	Full	#Cons	#Cmd	t_S	t_C	Mem	I/O	#SC	#FD	
analytics	SA/DA	4	53	1.94GB	78.9GB	10	21	10ms	84s	334MB	23.0GB	117.3m	84	[21, 79, 84]
nginx.sh			19			10	13	~0	1s	10.3MB	1.79GB			
…														
bio	DA/BI	6	872	24.3GB	114GB	17	71	51s	6720s	25.1GB	352GB	35.3m	79	[8, 42, 78]
bio.sh			23			11	14	10ms	176s	20.4MB	16.3GB			
data.sh			226			13	44	46s	3521s	25.1GB	287GB			
…														
ci-cd	CI/BS	21	2592	N/A		20	134	30ms	128s	461MB	2.35GB	3.5m	885	[15, 73]
lsmtest.sh			63			12	16	~0	30ms	332KB	1.69MB			
build.sh			20			10	8	~0	15s	131MB	566MB			
…														
covid	DA/DE	5	53	3.34GB	5.08GB	5	6	~0	67s	1011MB	80.6GB	14.3m	150	[90]
etcetera	MI/MI	2	193	N/A		20	43	20ms	71s	15.8MB	26.2GB	7.1m	141	[53, 59]
sieve.sh			45			15	21	20ms	71s	15.8MB	26.2GB			
try.sh			148			17	29	~0	30ms	1.65MB	1.05MB			
file-mod	AN/MI	5	41	4.35GB	39.2GB	11	10	99ms	235s	164MB	13.9GB	1.5m	61	[79, 81, 84]
encrypt.sh			11			10	6	~0	2s	2.82MB	5.10GB			
img-conv.sh			11			11	6	99ms	170s	145MB	3.83GB			
…														
inference	ML/DA	3	60	3.85GB	11.7GB	15	27	40ms	1586s	7.16GB	83.7GB	37.9m	81	[49, 92]
interact	SA/MI	2	96	11.1MB	14.9MB	17	20	33s	292s	224KB	5.58GB	154.0m	14	[19, 85]
ml	ML/DA	1	47	4.71GB	15.0GB	7	1	~0	1156s	7.87GB	41.6GB	14.9m	10	[71]
net	NW/SA	3	146	147KB	1.43MB	18	39	339ms	8s	42.5MB	135MB	3.4m	83	[63]
ping.sh			20			14	8	130ms	510ms	768KB	6.43MB			
…														
nlp	TP/ML	23	303	3k bks	115.9k bks	10	19	15s	851s	9.62MB	272GB	178.6m	526	[11]
oneliners	AN/TP	13	116	202MB	13.5GB	10	23	60ms	14s	199MB	3.77GB	4.5m	288	
spell.sh			11			6	7	~0	1s	8.38MB	523MB			[2]
top-n.sh			2			5	6	~0	960ms	8.25MB	408MB			[3]
uniq-ips.sh			2			4	3	~0	60ms	8.15MB	54.2MB			[58]
…														[80, 87]
pkg	CI/AN	2	43	110 pkgs	2.0k pkgs	16	22	5s	201s	572MB	15.3GB	35.2m	132	[9, 96]
pacaur.sh			29			14	19	5s	81s	490MB	14.6GB			
proginf.sh			14			11	6	10ms	120s	572MB	709MB			
rand	MI/MI	2	20	14.7MB	14.7MB	10	7	43s	934s	48.2MB	472GB	118.2m	30	[81]
repl	SA/MI	3	586	N/A		19	53	~0	24s	197MB	1.21GB	5.6m	79	[43, 79]
unixfun	MI/TP	36	70	599MB	59.1GB	4	12	~0	5s	9.80MB	5.71GB	944.0k	733	[5]
weather	DA/DE	2	74	893MB	146GB	11	20	260ms	958s	94.2MB	39.1GB	56.7m	50	[99]
web-search	MI/TP	7	82	833MB	8.61GB	16	39	11s	1343s	1.32GB	17.1GB	112.6m	174	[67]
Min		1	20	147KB	1.43MB	4	1	~0	5s	224KB	135MB	944.0k	10	
Mean		7.8	302.6	2.98GB	30.9GB	13.1	31.5	9s	815s	2.47GB	80.9GB	50.1m	200	
Median		3.5	78	991MB	10.2GB	13	21.5	79ms	218s	198MB	20.1GB	25.0m	83.5	
Max		36	2592	24.3GB	146GB	20	134	51s	6720s	25.1GB	472GB	178.6m	885	

3 Koala Design Overview

Tab. 1 summarizes Koala's full set of benchmarks and their characteristics. Koala offers 140 programs, analogous to but diverse from the earlier **`inference`** example (§2), grouped into sets that share features, sources, or inputs. This section focuses on the first three sets of columns: Identification characteristics (Col. 1) list each benchmark set's name and (a subset of) the programs it contains; Descriptions (Cols. 2–4) summarize application domains ($\mathcal{D}$), the number of programs in the set (#.sh), and the total lines of code (LoC); and Inputs (Col. 5) summarize the size of each input. Later sections discuss other characteristics.

Both programs and inputs are designed to be adaptable, so researchers can vary datasets or substitute components to evaluate other computations while leveraging Koala's diverse and well-understood programs. Beyond static programs, Koala also provides change sequences for each benchmark: curated edits and input updates that model incremental development (see §7.6 for an instance of their use for an incrementalization system).

Inputs: Before discussing specific programs and inputs, it is worth outlining the general structure of inputs, its goals, and the infrastructure deployed to ensure scalable and permanent storage. These are guided by the experience of several users [27, 40, 83, 95] over multiple years.

Koala comes with three input sizes for each of its benchmarks. Full inputs (`--full`) are either the original inputs the programs were designed to operate on (*e.g.*, **`weather`**, **`covid`**) or, for older programs, realistic large-scale inputs on which these programs would operate today (*e.g.*, **`oneliners`**, **`unixfun`**). Small inputs (`--small`) are subsets of the full inputs useful for smaller-scale characterizations. They range between 0.1–30% of the original input (with some exceptions, noted below) and are carefully truncated to still be meaningful and semantically valid—for example, genomics pipelines still operate on valid genome sequences seen in practice, rather than arbitrary strings terminated at arbitrary points. Minimal inputs (`--min`) are synthetic data created to quickly confirm correct setup and facilitate further automated configuration. These inputs aim at near-immediate results and, depending on the characteristics of the system and environment, are either downloaded or generated. Downloading, as detailed below, fetches inputs from cloud servers when high-bandwidth connectivity is available; generation operates repeatedly on benchmark-specific seeds, useful for environments with limited connectivity.

To meet key performance and availability requirements, inputs are hosted on infrastructure that combines two replication tiers—in addition to their source, *e.g.*, NOAA [66] or Wikipedia dumps [14]. The primary tier operates as Koala's default configuration and stores all inputs on a replicated storage cluster managed by Brown University. The cluster is connected via a 1 Gb/s switched Ethernet network with access to Internet1 and Internet2 via Brown's fiber-optic backbone, aimed at high (but not permanent) availability—ensuring that inputs are available for concurrent users of the Koala benchmarks. Additionally, infrastructure managed by Stevens Institute of Technology and UCLA hosts replicas of all inputs to offer additional availability.

The secondary tier ensures permanent availability via archival storage. Inputs (and the programs using them) are made available on Zenodo servers, split appropriately to comply with Zenodo's 50GB limit. While it does not offer the same bandwidth as the primary tier, it forms a transition path if input reconstruction becomes necessary—in the unlikely case that all university replicas become permanently unavailable.

Computational domains: Koala draws from several domains, representing a broad range of computations—including both classic workloads typical of shell programs and modern workloads found pervasively today.

Data analytics programs (DA, 11 programs across three sets) extract, transform, and summarize quantitative datasets such as temperature records, public transit data, and genomic information.

System administration programs (SA, nine programs across three sets) include typical system setup and maintenance tasks, system log manipulation, or software installation. Continuous integration and deployment workflows (CI, 23 programs across two sets) include standalone shell programs or ones that automate software builds, program analysis, and software testing. Machine-learning programs (ML, four programs across two sets) include scripts either implementing learning tasks or gluing together third-party components in modeling pipelines. One-off automation scripts (AN, 18 programs across two sets) automate various operations such as file encryption, media conversion, and one-off tasks such as spell-checking and content filtering. Networking programs (NW, three programs in one set) include scripts that interact with the system's networking stack, such as scanning for open ports. Other miscellaneous benchmarks (MI, 47 programs across four sets) consist of scripts that do not belong to any particular domain and perform a variety of tasks such as arbitrary transformations or non-deterministic computations.

Across and orthogonal to these domains, Koala combines several diverse styles of computations and tasks (after the slash in column $\mathcal{D}$). For example, two data-extraction sets (DE) summarize information from large data sources; four text-processing sets (TP) manipulate text in multi-stage transformation pipelines; one bioinformatics set (BI) processes data generally related to biology applications; one build script set (BS) includes a variety of software development build scripts; and three automation sets (AN) manage task execution.

Benchmark programs: Koala consists of 18 sets of programs, presented in alphabetical order.

The **`analytics`** set contains four programs that analyze log files to extract key events [79, 84]. Operations include filtering and summarization. They operate on 78.9GB of line-oriented logs—including TCP traffic, Nginx access logs, and ZMap scan data—all collected from real network traces, as well as logs from a ray-tracing system [17, 22, 25, 82]. The small inputs (1.94GB) include truncated versions of the same log and packet-capture files.

The **`bio`** set is six programs for processing genomic and transcriptomic data. One performs population genomics analysis [8, 78], and several implement key stages of the TERA-Seq platform [42] for processing and aligning RNA sequences. Inputs include a BAM genome sequencing file [36] and auxiliary data such as gene annotations, totaling 114GB. The scripts exhibit fan-out/fan-in structures, offer opportunities for code de-duplication, implement work-queue-like parallelism, and operate on large files. Small inputs (24.3GB) omit much of the optional auxiliary data.

The **`ci-cd`** set contains 21 build-related programs, including scripts for building eight applications—such as Lua, Memcached, Redis, and SQLite [15]—as well as the `makeself` utility, which creates self-extracting archives [73]. These programs are used in continuous integration and deployment pipelines and feature multiple dependencies without any data inputs. The `makeself` script executes the program's test suite and requires no inputs.

The **`covid`** set contains five programs that calculate statistics about the public transit activity of a large city during the COVID-19 pandemic [90]. These programs come in two versions, amenable to different optimization strategies: a version that uses typical Unix staples such as `cut`, `sort`, and `uniq`; and one written as a monolithic `awk` program. They operate on 5.08GB of CSV data about transit vehicle activity, and the small inputs cover a 3.34GB subset.

The **`etcetera`** set contains two programs that perform computations that are difficult to statically reason about, or complicate dynamic analysis due to their use of self-modifying code and reflection. The first program dynamically generates code, which it then runs inside an isolated environment using `unshare` and `chroot` [50]. The second program computes Eratosthenes' sieve using a recursive shell function [59]. Neither program requires any inputs.

The **`file-mod`** set consists of five programs that automate file-level transformations, including compression, encryption, and format conversion. Two of the programs operate on packet

capture files collected from publicly available datasets [64], compressing and encrypting them using `openssl` [81]; the full inputs total 39.2GB. The other three perform media format conversions [79, 84], reading from and writing to the filesystem in tight loops that process hundreds of image and audio files (4.4GB). The small version of the benchmark set uses a reduced dataset with 4.35GB total of text and media files.

The **`inference`** set contains three programs that perform inference tasks on media files using large foundational models [48, 88, 104]. These include image captioning [92], music playlist generation via embedding interpolation [49], and sequential segmentation and classification of hieroglyphic images using SAM [48] and a custom classifier. These benchmarks process 9.3GB of image and audio data. The deep-learning models total 2.4GB in size, for 11.7GB of inputs overall. The benchmarks perform model serving using external runtimes [68, 70] and interface with them via custom wrappers [100]. The small version operates on a subset (3.85GB) of the original images and music using the same back-end models.

The **`interact`** set contains two programs that accept interaction throughout their execution. The first program is an interactive installation for the `ohmyzsh` framework [19] and the second is an excerpt from a text-based game, which accepts user commands to simulate navigation inside a virtual environment [85]. The first program requires no inputs, while the second operates on a text file containing a sequence of key presses (5 million for full inputs, 1 million for small).

The **`ml`** set consists of a typical machine-learning workload comprising multiple stages written using the Scikit-Learn library [71]. It is a decomposition of a monolithic Python program into a shell program that orchestrates distinct steps for data ingestion, learning, inference, classification, and evaluation on 15GB of inputs. The small version uses a subset of the same input (4.71GB).

The **`net`** set includes three programs that interact with the network [63]. One program performs network scanning using `nmap` [77], another performs a ping sweep over several IP addresses, and a third configures the system's `iptables` firewall rules [89]. The first program requires no inputs, while the second and third operate on lists of randomly generated IP addresses (full version uses 5,000 and 100,000 addresses, respectively; small version uses 500 and 10,000).

The **`nlp`** set contains 23 scripts implementing natural language processing techniques, drawn from the "Unix for Poets" NLP tutorial [11]. Most scripts consist of 1–2 lines, and can be combined in sequential operation. The input dataset contains over 115,000 ASCII books from Project Gutenberg [37], and 3,000 books (1.42GB) for the small inputs.

The **`oneliners`** set contains 13 shell pipelines drawn from various sources across the academic and popular literature [2, 3, 11, 21, 58, 80, 87]. Some are Unix classics [2, 3, 80], highlighting the Unix philosophy, embodied by the shell; others are more recent [21, 58, 87], applying the same principles to modern workloads. They make extensive and, at times, complex use of streaming constructs, applying maps, filters, reductions, stream duplication, and window operators. Their inputs are shared between script subsets and combine books from Project Gutenberg, commands available in `PATH`, and packages from the `apt` repositories (13.5GB full, 202MB small).

The **`pkg`** set consists of two programs: one automates the installation of packages from the Arch User Repository (AUR) [55], and the other applies a static analysis tool to extract permissions from the Node.js package manager (npm) repository [65, 96]. Its input consists of two lists—195 AUR package names and 1,768 npm package names (about 2.0k packages total). The benchmark downloads, builds, and installs the AUR packages in a loop, and analyzes the npm packages to infer their permissions. The small version of this benchmark includes the first 10 and 100 packages from each list (110 packages total).

The **`rand`** set contains two programs that generate or randomize data: one generates a set of random strings by probing `/dev/urandom`, and another shuffles lines in a large text file before sampling a subset of them to create ten smaller files [81]. The first script takes no data inputs, while

the second operates on a text file of two million English names [93] (14.7MB for both small and full inputs). The full version generates 50 million 32-byte random strings for the first program, and 100,000 sets of 10 names each for the second; the small version generates 500,000 24-byte random strings and 10,000 sets of 10 names each.

The **repl** set contains three standalone programs that resemble live shell sessions. Two perform security auditing for vulnerabilities and misconfigurations, and another replays a development workflow over a large Git repository. These scripts include non-trivial filesystem access patterns, with the repository workflow being metadata-heavy. They use no inputs.

The **unixfun** set contains 36 programs solving the Bell Labs 50-year Unix anniversary challenge [5]. They mimic classic Unix text-processing computations, often short ones that eliminate the vast majority of the input using a `head` or `tail`. They operate on inflated inputs (59.1GB) created by duplicating the original inputs (599MB), while maintaining the expected structure.

The **weather** set consists of two program phases calculating statistics on historical temperature data from the Hadoop book [99], with the second one performing some additional analysis and recreating a weather diagram presented by Edward Tufte [91]. Some of these phases correspond to MapReduce and Spark computations exemplifying large-scale data processing—not part of Koala, but useful for comparisons of shell-acceleration systems targeting similar or comparable scalability goals. These phases operate on large inputs (146GB) collected from NOAA spanning multiple years [66]. The small version corresponds to a subset of temperatures from 2015 (893MB).

The **web-search** set contains several programs from the initial (non-distributed) assignment of an advanced course on distributed computing systems, implementing web crawling, indexing, and querying as a POSIX-shell streaming program, using complex streaming operators—including duplication, shifting, windows, and dataflow cycles. Inputs consist of a Wikipedia snapshot (8.61GB), or a subset for the small version (833MB).

Extensions: Koala encourages extensions in several ways, in order to facilitate evaluation of systems that target different computational aspects of shell programs. Such extensions can be used to evaluate performance-optimization systems that target different computational aspects beyond parallel and distributed execution. One example is incrementalization [16, 32], accelerating the re-execution of modified programs by avoiding recomputations of program fragments that remain unchanged. Another example is bolt-on reactivity, accelerating the re-execution when inputs, rather than programs, change. Yet another example is notebook-style computations *i.e.*, systems that allow interactive refinement of exploratory computations [47].

The current version of Koala includes extensions targeting incrementalization and reactivity. For incrementalization, it includes a series of modifications across all Koala benchmarks for which it was possible to create incremental-development scenarios given the information currently available (*e.g.*, GitHub commits). Each modification corresponds to an addition, deletion, or in-place modification in one of the Koala programs. Each step towards the final version of the program is categorized based on the type of modification (addition, deletion, modification, or a combination thereof) and the rationale behind the change: (B) behavior change, (C) wrong command fix, (F) wrong flag fix, (E) exploration, (S) summarization, (O) optimization, (L) LLM assistance, (R) replacement, (I) input update, (D) debugging, (A) aggregation, or (V) visualization.

The set of benchmarks that come with modifications include: **analytics**, (11E 6A R 5D S 6F O O E S), **bio** (I B I D 2E), **covid** (4E), **file-mod** (3L D O O), **inference** (O 2F I A F V 2C 2D 2D L O), **nlp** (3E 3B), **oneliners** (S 3D E 2B), **unixfun** (2E E 2E), and **weather** (2E).

Similarly, for reactivity, Koala supports various inputs deltas—in the form of both truncations and inflations. Such deltas capture changes to a program's input data rather than the program itself.

This set of changes in program and inputs is offered with this version of Koala, and is used in the application of Koala to state-of-the-art systems (§7).

Principal component analysis: Before diving into various static and dynamic characteristics of the Koala suite, it is worth getting a sense of its diversity characteristics at a high level. Fig. 2 shows this diversity using Principal Component Analysis (PCA) [1] on two distinct representations of each benchmark detailed below.

PCA maps high-dimensional data to a lower-dimensional space that preserves structural differences for easier comparison and visualization. It computes weights for each of the n original dimensions to create k (where $k < n$) new and uncorrelated composite dimensions, each linear combinations of the original ones.

The top row of Fig. 2 visualizes benchmarks projected on this reduced space based on their static and dynamic characteristics (§5–6). The spread of the benchmarks across the principal components suggests that the suite covers a broad and non-overlapping range of syntactic and behavioral profiles. The bottom row of Fig. 2 visualizes each benchmark in a space based on dense vector representations that capture high-level program structure and logic. These embeddings, generated using OpenAI's `text-embedding-3-large` model [69], capture information targeting diversity at the semantic and syntactic level [74, 94]. Well-distributed across the projected space, the results also suggest substantial diversity in both syntax and semantics.

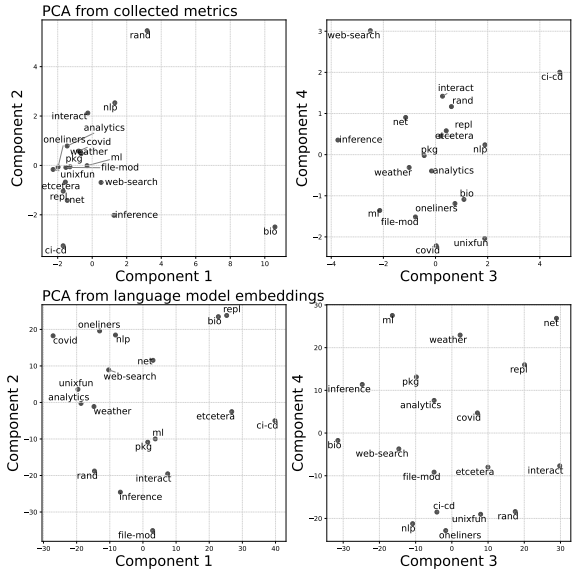

Fig. 2. Principal component analysis results. Top: PCA plot resulting from the static and dynamic analysis of each benchmark. Bottom: PCA plot resulting from Koala source code embeddings using a language model [69].

4 Infrastructure and Configuration

Each Koala benchmark set adheres to a specification that defines installation dependencies, input configuration, benchmark execution, and validation of the final results. This specification, shown in Fig. 3, includes five infrastructure scripts: (1) `install.sh` sets up dependencies required by the benchmark; (2) `fetch.sh` downloads (or generates, §3) and, if necessary, pre-processes inputs, accepting an argument that specifies their size; (3) `execute.sh` executes benchmark programs, collecting basic information such as execution time and resources; and (4) `validate.sh` confirms the correctness of the benchmark's output by hashing the output of each script and comparing it to a known-good hash, sometimes after applying transformations

```
sample-benchmark
├── scripts       # Benchmark scripts
│   ├── a.sh
│   ├── b.sh
│   └── ...sh
├── install.sh    # Dependency installation
├── fetch.sh      # Input data download/preparation
├── execute.sh    # Script execution
├── validate.sh   # Hash generation & verification
└── clean.sh      # Input and output file cleanup
```

Fig. 3. Benchmark structure. Each benchmark includes five support scripts and a `scripts/` directory containing the source code of each benchmark. A top-level `main` driver executes all support scripts for one or all benchmarks.

to account for light non-determinism or slight formatting differences (*e.g.*, a compiled artifact with an embedded timestamp). Finally, (5) `clean.sh` removes inputs, outputs, and temporary files created during the benchmark's execution. These scripts avoid redundant work—*e.g.*, dependency installation or input generation are skipped if possible, unless forced (`-f`).

Koala also includes a top-level driver (`main.sh`) that executes the five scripts in sequence. It accepts several configuration parameters, which it then forwards to each script. For example, a `KOALA_SHELL` parameter configures the executing shell interpreter *e.g.*, `bash`, `zsh`, or a path to an executable, defaulting to `sh`.

Koala's structure aims to facilitate quick collection of preliminary results, not exhaustive coverage of all possible needs. To accommodate a wide range of systems, it is kept minimal—inviting users to modify it to fit their needs.

Containerization: Koala provides optional infrastructure for running benchmarks in an isolated environment. It includes a Dockerfile that builds a Debian-based container, installs essential dependencies such as `git` and `sudo`, and fetches the suite. A volume is shared with the host system, simplifying access to inputs and outputs.

It also provides support for dynamically generating container images tailored to each benchmark. These images are self-contained and ephemeral: their Dockerfile contains a `CMD` command executing all infrastructure scripts via `main.sh`.

To avoid complications with permission changes or system-wide modifications, Koala does not require elevated privileges (except for `fetch.sh`, which installs software dependencies) and avoids privileged commands such as `sudo` and `setuid`. For scripts that modify global system state (*e.g.*, **`net`**'s `iptables` configuration), Koala runs them inside a namespace using `unshare` to avoid affecting the host system.

Integration effort: The effort required to add a new benchmark to Koala depends on its complexity, input size, and correctness constraints. It typically involves five steps: (1) identifying and encoding dependencies in `install.sh`; (2) preparing input datasets in multiple sizes—full, small, and min—via `fetch.sh`; (3) including scripts for automated execution in `execute.sh`; (4) defining output validation logic in `validate.sh`; and (5) specifying cleanup steps in `clean.sh`. Integrating the current set of benchmarks ranged between 10–80 person-hours per benchmark (Tab. below; *P* column). Small and self-contained benchmarks such as **`oneliners`**, **`unixfun`**, and **`nlp`** each took about 10–12 hours; more complex or data-heavy benchmarks such as **`file-mod`**, **`bio`**, and **`ci-cd`** took about 60–80 hours due to their size, number of dependencies, and nuanced verification which went beyond hash comparison. Most of the time for this release of Koala was spent collecting the new benchmark programs and confirming their correctness under various environments and configurations.

Benchmark	*P* (h)	*A* (h)
`analytics`	40	2
`bio`	60	5
`ci-cd`	75	3
`covid`	65	3
`etcetera`	10	0.5
`file-mod`	60	0.5
`inference`	35	1
`interact`	30	1
`ml`	30	1
`net`	15	0.5
`nlp`	10	1
`oneliners`	10	2
`pkg`	30	2
`rand`	15	0.5
`repl`	25	0.5
`unixfun`	10	1.5
`weather`	30	1
`web-search`	40	3
Total	590	29

Evaluating systems with Koala requires varying levels of effort depending on their design goals, degree of automation, and artifact maturity (Tab. on the right; *A* column). The total effort across all systems was approximately 29 person-hours and ranged between 0.5–5 hours per benchmark. For the Shark and `parallel` systems, this involved manual changes to the benchmark scripts. For Shark, modifications typically included identifying parallelizable regions in loops and applying background execution with `&` and `wait`,

along with the removal of unnecessary `cat` commands and the use of tee to preserve I/O semantics (*e.g.*, in **bio**, **analytics**, and **file-mod**). In some benchmarks, such as **ci-cd**, **repl**, and **unixfun**, changes spanned tens of lines due to multiple independent compilation or execution steps. For `parallel`, adaptations involved wrapping loops or commands using the `parallel` utility. This process required changes in the order of 1–3 LoC in benchmarks with stateless pipelines or simple loops (*e.g.*, **pkg** and **nlp**), but was more intrusive in cases like **covid**, **analytics**, and **repl**, where it required identifying parallelizable regions, exporting computations as shell functions, and occasionally introducing temporary files to manage intermediate data. These modifications reflect the intended use and capabilities of each system rather than any requirement imposed by Koala, and are documented in two public repositories.[1] In contrast, `zsh`, PaSh, *hS*, and Incr were applied without any manual changes.

5 Syntactic Characterization and Analysis

Contrary to single-language environments, the shell often combines components in multiple languages. Koala thus distinguishes between the characteristics of (and resources spent on, see §6) the shell portion of each benchmark, versus those of components called into by the shell and which often implement the core of a computation (Tab. 1, cols. 6–7).

Methodology: We use `libdash` [54] to parse and analyze both portions using SMOOSH's abstract syntax definition for the POSIX shell [30]: for the shell portion, we count the total occurrences of every AST node; for the command portion, we analyze only AST nodes counting commands, built-ins, and functions—yielding conservative results that exclude dynamic constructs (*e.g.*, `$CMD arg`) but avoid conflating static structure with repeated runtime executions.

Language features: Fig. 4 summarizes the occurrences of each construct across Koala. Each cell shows the number of times a syntactic construct (vertical axis) occurs in a benchmark (horizontal axis). AST nodes that overlap with more specific nodes or that do not correspond to linguistic features (*e.g.*, escaped characters) are excluded. Overall, Koala employs all the syntactic constructs of the POSIX shell, in varied combinations that reflect its various styles of computation.

Koala covers several key shell characteristics worth mentioning. Most sets employ pipelines, shell constructs of the form `a | b` that pass the output of one command (`a`) as input to the next (`b`). Pipelines are an important shell feature, optimized by several data-parallel systems. Two sets use operators `&` and `wait` for explicit parallelism—a feature relevant to performance-oriented shell systems. Two sets use sub-shells, *e.g.*, `(echo hi)`, and several sets use control constructs such as loops and conditionals. Several sets use function definitions, and nearly all use variables, assignments, and stream or file redirections; other kinds of redirections occur less frequently. Many sets use command substitution, *e.g.*, `echo hi $(whoami)`.

Fig. 4 also highlights a key Koala characteristic: no two benchmarks exhibit the same distribution of syntactic constructs. Koala represents well the shell's most interesting features—*e.g.*, external command invocations, pipelines, the background operator, subshells, expansion, and redirection. In combination, these features are essential in making the shell the uniquely powerful and complex platform that it is. Different styles of scripts employ these features in different combinations and proportions—and Koala aims to capture this variation. The proportion of these features aligns with shell usage studies [20], indicating that the distribution of Koala's shell features corresponds to current trends and is representative of the programs found in the real world.

Component features: Fig. 5 shows the frequency of all Koala commands occurring at least 30 times, excluding benchmark-specific functions and binaries. Koala contains many of the most common commands found in the wild: `echo` tops the list with 441 occurrences; `cat`, `grep`, `sort`,

[1] https://github.com/kbensh/koala-shark and https://github.com/kbensh/koala-parallel

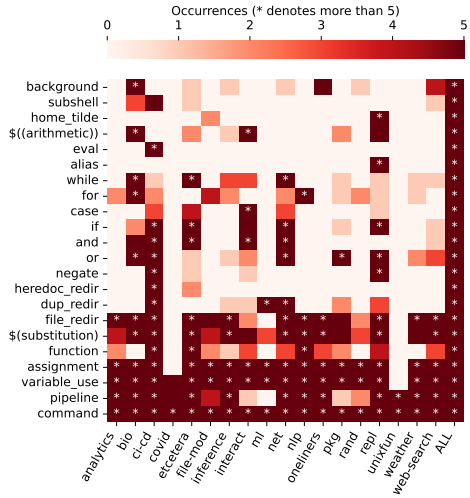

Fig. 4. Syntactic characteristics. Cells show the frequency of the syntactic construct (y-axis) in a benchmark (x-axis), up to five occurrences to maintain clarity of absences (zero counts). Stars denote more than five occurrences.

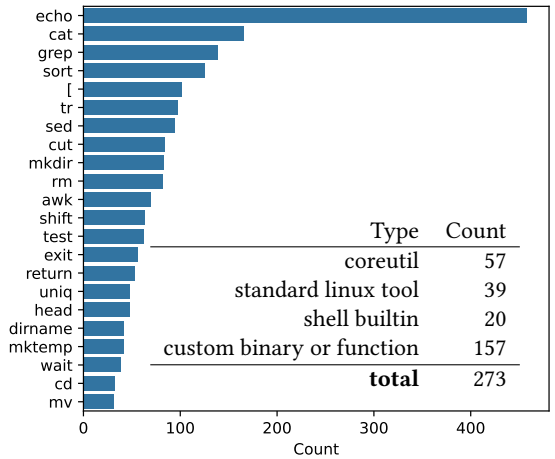

Fig. 5. Command occurrences. The chart shows the frequency of all Koala commands occurring at least 30 times. The table categorizes the unique commands counted across the entire suite.

and `tr` count about half of that; other commands exhibit lower frequencies. These results broadly match studies of commands in the wild [20], except for path-manipulation commands as most Koala paths are static.

The table also groups Koala's distinct commands into four classes. The most common are GNU Coreutils (*e.g.*, `echo` and `cat`), shell built-ins (*e.g.*, `cd` and `return`), and standard Linux tools (*e.g.*, `grep`, `sed`, and `awk`). While the first three classes account for the vast majority of Koala's commands by frequency, custom binaries and functions account for 157 commands, or 58% of the unique command set.

6 Dynamic Characterization and Analysis

Beyond its syntactic diversity, Koala also exhibits diverse dynamic characteristics. Exploring this diversity involves executing benchmarks to extract dynamic features such as CPU time, memory consumption, IO characteristics, and interaction with the broader environment (Tab. 1's cols. 8–13).

Methodology: To extract these features, we collect several metrics while executing benchmarks on a machine with 32GB RAM, an 8-core 3.80GHz Ryzen 7 9700X CPU utilizing hyper-threading, and a 1TB NVMe SSD. Unless otherwise noted, all characterization uses each benchmark's `--small` input tier, which preserves realistic, semantically valid inputs while keeping executions tractable for repeated measurement and analysis. The shell interpreter is set as `bash --posix`. We measure CPU time and IO by probing `/proc/<pid>/{stat,io}` after the target process exits; and wall-clock time and memory use by calling Python's `time.perf_counter` and `psutil` at 0.01-second intervals. We aggregate results by set, summing timing and IO statistics for each program in that set. We take the maximum high-water-mark memory usage to compute a single set of results per benchmark.

Overview: Fig. 6 presents results both overall and with respect to input sizes. The plots form two coarse groups (the x-axis always lists benchmarks): (1) the top two plots (left and right) show total execution time—left is the ratio of CPU time to wall-clock time, right is the total IO per second

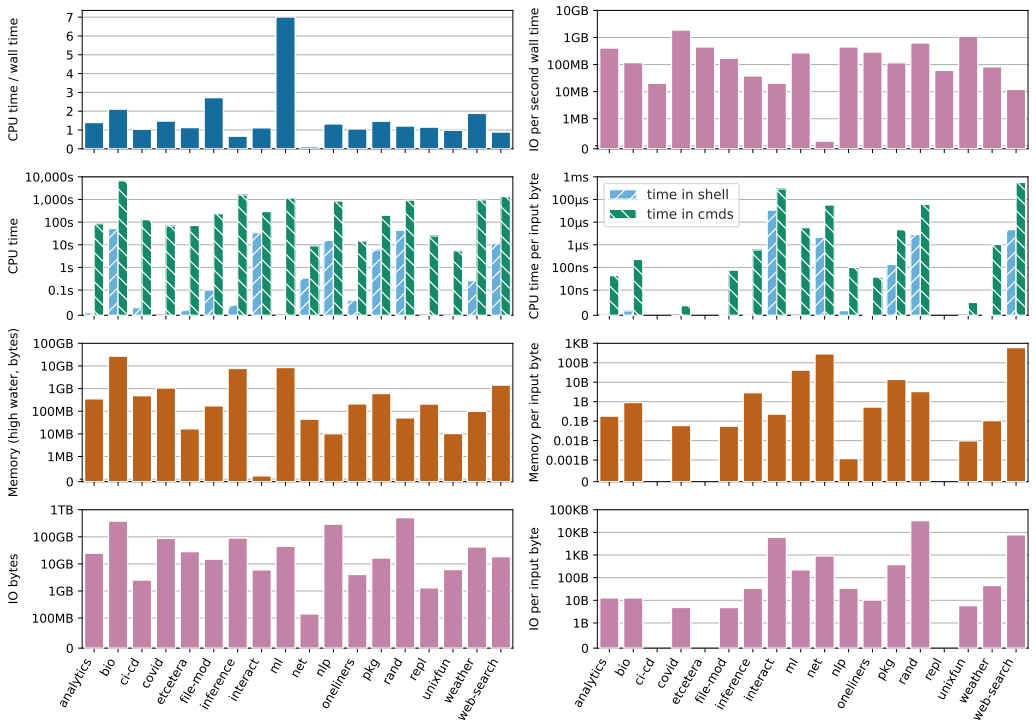

Fig. 6. Dynamic characteristics. The top two plots contextualize the rest: (left) ratio of CPU time to wall-clock time; (right) ratio of IO time to wall-clock time. The bottom three plots (rows 2, 3, 4) show CPU time, Memory, and IO: (left) absolute; (right) normalized over benchmark input. All y-axes except top left are symmetric log scale: values between 0 and the first tick are linearly scaled, and the next are logarithmic.

of wall-clock; (2) the next six plots (rows 2, 3, 4) show CPU time, high-water-mark memory, and IO—left is absolute, right is per input byte.

Absolute characteristics: As these characteristics vary by orders of magnitude, these plots use a symmetric log scale: values between zero and the first tick are linearly scaled, while the rest are logarithmic. The CPU-to-wall-clock ratio plot (Fig. 6, row 1, left) shows that Koala benchmarks exhibit a wide range of internal parallelism, from highly parallel (*e.g.*, **`ml`** with a ratio of close to 7) to ones that spend most of their time waiting for IO (*e.g.*, **`net`**, with a ratio less than 0.1). The CPU-time plot (Fig. 6, row 2, left) shows that the benchmark runtime varies from seconds (*e.g.*, **`unixfun`**) to hours (*e.g.*, **`bio`**). CPU time is split between time spent in the shell versus commands: here, too, Koala demonstrates significant diversity—ranging from a mix of shell execution and commands (*e.g.*, **`nlp`**) to command-dominated workloads (*e.g.*, **`bio`** and **`inference`**).

Similarly, the memory plot (Fig. 6, row 3, left) shows the diversity of memory intensiveness, varying between 224KB (*e.g.*, **`interact`**) and 25.1GB (*e.g.*, **`bio`**). Likewise, the IO plot (row 4, left) shows that Koala exhibits varying degrees of IO-heaviness, ranging from 135MB of IO (*e.g.*, **`net`**) to 472GB (*e.g.*, **`rand`**).

Tab. 1's last few rows (pg. 5) offer additional context: shell and command time range between 0–52s and 5.4–6720.9s, respectively (averages: 9.1s and 815.9s), memory consumption ranges between 224KB–25.1GB (average: 2.47GB), and IO between 135MB–472GB (average: 80.9GB).

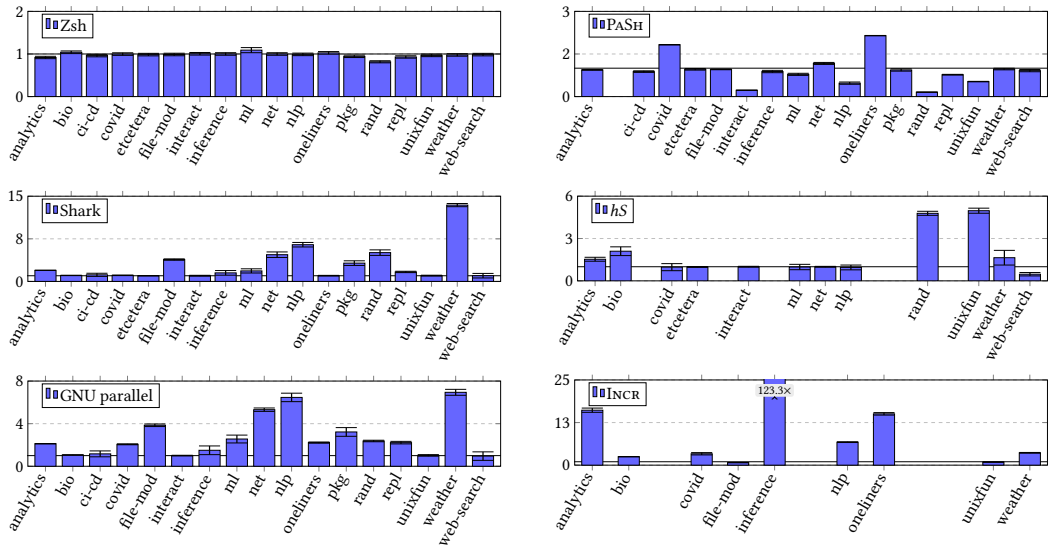

Fig. 7. Speedups applying zsh, Shark, GNU `parallel`, PaSh, hS, and Incr. Relative average speedup on the Koala benchmarks. Baseline is the original runtime in a single-threaded bash shell with the `--posix` flag enabled.

Normalized characteristics: Normalized plots (rows 2–4, right) offer a sense of these characteristics normalized by each benchmark's input size, important because of their correlation with input size, which varies widely (0–146GB). This normalization is noticeable with **`bio`** and **`covid`**: these benchmarks have substantial absolute CPU times (on left), but their normalized CPU times (on right) are moderate—reflecting the fact that they are long-running mainly due to the volume of data they process, not the intensity of their computations. On the other hand, **`pkg`** is computationally intense, despite its middling absolute execution time. These characteristics do not apply to benchmarks with no inputs (*e.g.*, **`ci-cd`**, **`etcetera`**, and **`repl`**), which have no bars in the normalized plots.

Overall, across the three dimensions (CPU, row 2; memory, row 3; and IO, row 4), Koala enjoys significant diversity—multiple metrics vary by several orders of magnitude.

7 Applying Koala to Optimization Systems

This section summarizes the process and results of applying six optimization systems (§2) on the Koala benchmarks: The `zsh` interpreter [24], Shark [4], GNU `parallel` [86], PaSh [95], *hS* [52], and Incr [102] achieve performance gains on different subsets of Koala.

Hardware & software setup: All experiments in this section were executed on an AWS `c6i.4xlarge` instance with 32GB of RAM and a 16-core 3.5 GHz CPU running Ubuntu 24.04.1 and `bash` v5.2.21 with `--posix` enabled.

Adoption effort: The six systems require different levels of effort to use Koala (see 4), depending on their needs (*e.g.*, automation, inputs), characteristics, and goals (§2). GNU `parallel` required modifying benchmarks to export parallelizable pipelines and **`for`**-loops as functions passed to `parallel`. Shark required similar transformations guided by its optimizations. The `zsh` interpreter and PaSh, *hS*, and Incr, operating as drop-in shell replacements, simply set `KOALA_SHELL`.

7.1 Applying Koala to Zsh

To understand whether Koala aids the characterization of `zsh` [24], we run all benchmarks using `zsh` under its `sh` emulation mode and compare their runtime against `bash`.

Methodology: We run all benchmarks using `zsh` v5.9, comparing their runtime against `bash` v5.2.21 under its `sh` emulation mode.

Results: Under `zsh`, all benchmarks in the suite complete successfully. The `zsh` shell interpreter achieves a 0.16% average slowdown compared to `bash`. This confirms that `zsh` does not introduce significant slowdown relative to `bash` when applied to a variety of real-world shell programs.

Takeaway: Koala confirms that `zsh` risks no significant slowdown over `bash` on shell programs.

7.2 Applying Koala to Shark

To understand whether Koala aids the characterization of Shark [4], we manually apply its transformations as described in the Shark paper across several Koala programs.

Methodology: The Shark paper [4] describes several potential syntactic transformations, which we apply manually across all Koala benchmarks. We then execute the original and optimized versions to characterize the performance of the modified benchmarks.

Results: Shark achieves significant performance gains across Koala, ranging between 1.01–13.43×, depending on benchmark characteristics. Benchmarks that involve trivially parallelizable loop iterations see major speedups—*e.g.*, Shark improves the **`nlp`** and **`weather`** benchmarks by 6.46× and 13.43×, respectively. Scripts with sequential operations, such as those found in **`ci-cd`**, exhibit less pronounced improvements—at times only marginal gains of 1.16×.

Shark accelerates most benchmarks, though benchmarks with highly interdependent operations that already use pipelines see more limited gains. For example, **`covid`**, **`oneliners`**, **`bio`**, and **`unixfun`** offer fewer opportunities for Shark-style optimizations, which result in speedups between 1.01–1.06×. The **`web-search`** benchmark sees the least improvement (1.01×) as it already runs the three n-gram calculations in parallel, making Shark obsolete. Shark's optimizations centered around command invocation do not result in significant performance benefits, speeding up scripts where only such optimization opportunities were present (*e.g.*, web-search) by 1.01× on average. In contrast, optimizations that eliminate temporary files used for intermediate storage can offer more substantial speedups, but they also introduce significant complexity, as pipes require more careful coordination between producers and consumers.

Takeaway: Koala confirms that Shark's optimizations are effective in scripts that involve operations on multiple inputs and independent commands, and less effective for I/O-heavy scripts or scripts that are not easily parallelizable. Its parallelization and pipeline optimizations offer significant speedups, up to 13.43×.

7.3 Applying Koala to GNU `parallel`

To understand whether Koala aids the characterization of GNU `parallel` [86], we apply it to several Koala programs focusing on natural candidates for such parallel execution.

Methodology: We first identify parallelizable regions within each Koala program and rewrite them to invoke `parallel`. We aimed for a reasonable use of `parallel`, modifying only parallelizable pipelines that each take under an hour to identify and rewrite—typically ones with (1) independently processed input files, and (2) efficient segment-based processing that requires minimal or no synchronization and aggregation. The latter often takes advantage of GNU `parallel`'s `--pipe` to parallelize input stream processing.

Results: GNU `parallel` achieves substantial speedups in benchmarks that involve I/O-bound operations or are trivially parallelizable. For instance, **`file-mod`** involves converting multiple media files concurrently, resulting in GNU `parallel` speeding it up by 3.84× and fully utilizing all available CPU cores. Similarly, **`nlp`** processes multiple data files independently, resulting in a speedup of 6.46×. GNU `parallel` speedups are less pronounced for scripts that do not have these features, such as those in **`ci-cd`** and **`repl`** that result in 1.16× and 0.95× respectively. These additionally either already use parallelism internally in their command invocations (compiling multiple files) or include commands that do not benefit from `parallel`'s parallelization model, such as `git` or `find`.

A few cases do not yield speedups. For example, the **`unixfun`** benchmark involves pipeline stages dependent on the output of previous stages, limiting `parallel`'s approach: the interdependent nature of these stages prevented GNU `parallel` from fully exploiting parallelism, resulting in an average speedup of 1.02×.

Takeaway: Koala confirms that GNU `parallel` can accelerate I/O-bound tasks and loops with no interdependencies—*e.g.*, **`file-mod`**—achieving average speedups of 2.6×. Limited acceleration comes from scripts with sequential operations or ones that require sophisticated splitting and aggregation (*e.g.*, **`unixfun`** and **`web-search`**).

7.4 Applying Koala to PaSh

We apply PaSh [46] to Koala to understand its ability to characterize PaSh's acceleration capabilities.

Methodology: We use the latest version of PaSh (commit 0d0a563) and set `KOALA_SHELL` to `./pa.sh --width 4`, *i.e.*, configuring the parallelization degree to 4×. We do not replace script fragments via `alias` and **`function`** constructs that can be annotated with additional parallelizability information; this would allow PaSh to extract additional speedup but would result in significantly more work. PaSh fails when executing the **`bio`** benchmark.

Results: PaSh's command-aware parallelization strategy achieves significant speedups in scripts with multi-stage pipelines or **`for`** -loops with no data dependencies across iterations. For example, it achieves speedups of 2.14× and 1.82× on **`oneliners`** and **`covid`** respectively.

Naturally, benchmarks or benchmark fragments for which PaSh had no annotations, and thus does not parallelize, see no speedups—*e.g.*, **`pkg`**, **`bio`**, and **`file-mod`**. Additionally, PaSh does not speed up scripts that include no syntactic constructs it can parallelize—*e.g.*, **`repl`** and **`ci-cd`** benchmarks, which do not include pipelines or parallelizable **`for`** -loops.

Takeaway: Koala reveals that PaSh can deliver substantial speedups for scripts that fall within its parallelization domain. However, its effectiveness depends on the availability of command annotations and the amenability of programs to the constructs PaSh can operate on.

7.5 Applying Koala to hS

To assess whether Koala aids the characterization of *hS* [52], we apply it to Koala, focusing on programs likely to benefit from out-of-order execution.

Methodology: As *hS* is currently under development, we use an early-stage *hS* prototype shared by its authors, configured via the `KOALA_SHELL` variable.

Not all benchmarks are executable by the current version of *hS*. For the following benchmark sets, *hS* succeeded on only a subset of the scripts: **`analytics`**, **`bio`**, **`ml`**, **`nlp`**, **`unixfun`**, **`weather`**, and **`web-search`**. The following benchmark sets as a whole either fail or produce incorrect results with *hS*, and are thus omitted entirely: **`ci-cd`**, **`file-mod`**, **`llm`**, **`nlp`**, **`oneliners`**, **`pkg`**, and **`repl`**.

Results: *hS* achieves significant speedups on scripts that include syntactic regions that have no dependencies between them. Speedups range between 1.5–4.97×, with **analytics** and **unixfun** at the two ends of this range. The **weather** and **bio** benchmarks also achieve significant speedups.

Scripts that involve dependencies between stages do not benefit from *hS*. Such slowdowns range from 0.97× in **covid** to 0.47× in **websearch**, averaging 0.72× across all benchmarks. They can be attributed to constant costs stemming from *hS*'s speculative execution, related to the environment isolation and tracing, which allows *hS* to roll back effects in cases of misprediction.

Takeaway: Koala reveals that *hS* can provide benefits that are significant, especially when computations feature independent stages. Typical computations, however, offer a mix of out-of-order optimization opportunities, at times masked by overheads in the *hS* prototype.

7.6 Applying Koala to Incr

We apply Incr [102] to Koala to understand whether it can characterize Incr's benefits for script re-execution. We leverage Koala's change sequences (§3) to showcase the benefits and limitations of incremental execution under real-world development scenarios and report Incr's average speedups across all modifications per benchmark.

Methodology: As Incr is currently under development, we use an early-stage Incr prototype shared by its authors, configured via the `KOALA_SHELL` variable.

Results: Incr achieves significant speedups on re-execution across all benchmarks, ranging between 2.48× and 123.29×, with an average speedup of 23.41×.

The **inference** and **analytics** benchmarks see the highest speedups of 123.29× and 16.12×, respectively: they either involve long-running computations skipped entirely on incremental re-execution, or include many modifications that compound incrementalization gains. Computations expressed in a few monolithic pipelines of 1–2 stages limit opportunities for incrementalization (*e.g.*, **bio** and **covid**). The **file-mod** and **unixfun** benchmarks have a 0.74× and 0.90× slowdown, respectively, due to Incr's overheads stemming from effect tracking. These benchmarks include tight **for** -loops with short-running commands inside them, making Incr's fixed overhead over all command executions compound over many iterations.

Takeaway: Koala reveals that Incr can deliver substantial speedups on re-execution across a variety of scenarios. Short-running commands inside tight loops can limit Incr's effectiveness due to its constant overheads.

8 Related Work

Benchmark suites: Progress in research depends on apples-to-apples comparisons—and in computer science, that often means open and reusable benchmark suites. Widely cited benchmark suites in memory-managed environments [7], database transactions [75], parallel processing [6], other areas [12, 39, 45] offer good examples of their widespread applicability and use. Similar to these suites, Koala comes with additional support and is expected to release improvements every few years; unlike them, it targets performance-oriented systems for the shell—an area that has not yet enjoyed the existence of a systematic benchmark suite.

The DaCapo [7] and Gabriel [26] benchmarks offer particularly good parallels, as they focus on programming environments that did not have a benchmark suite before their release—like these benchmarks, Koala fills a gap.

Shell studies and datasets: Recent studies collect shell scripts or fragments of scripts—typically in Bash—to analyze their source, understand their properties, and extract key insights. Examples of such studies include the use of Bash in the wild [20], the characteristics of build scripts in Linux distributions [44], and the use of aliases and shell customization in the wild [81], cybersecurity

training [97], and interactive coding [103]. These studies do not aim at (creating collections for) system evaluation within environments that execute, compose, or manage opaque components. In contrast, Koala aims to help characterize systems and tools that optimize shell programs. Koala offers curated end-to-end programs with known inputs, dependencies, and behaviors that stress both the shell and the underlying commands.

Shell test and correctness suites: A few test suites focus on evaluating the correctness of shell implementations and their conformance to certain standards. The POSIX test suite [35] is a key resource for validating compliance of a shell implementation with the POSIX standard. The *Smoosh* test suite [30] refines and complements the POSIX test suite. The Modernish shell library/polyfill has a sophisticated diagnostic routine that amounts to shell tests [18]. There are also a variety of testing frameworks designed for automated testing of shell scripts [34, 62, 76, 98]. All of these suites focus on testing functionality rather than characterizing systems within environments that include opaque components, and are thus distinct from and complementary to Koala.

Shell microbenchmarks: Several benchmark suites target the evaluation of specific shell-language constructs via microbenchmarks. `ShellBench` [61] provides a collection of small scripts (ranging from 8–95 lines of code) designed to stress individual shell features—*e.g.*, variable substitution, expansion, and subshell creation. Similarly, `zsh-bench` [72], the Oils benchmarks [10], and UnixBench [56] focus on isolated performance characteristics—*e.g.*, interactive shell behavior or command invocation times. In contrast, Koala offers larger, more diverse, whole-program benchmarks (140 programs across 18 sets, spanning 20–2592 LoC per set) that perform end-to-end computations, operate over large datasets, and involve substantial work outside the shell interpreter itself, and across many heterogeneous commands.

Performance properties and characterization: Prior research on benchmark sets often discusses language-specific properties about the programs involved—for example, the impact of garbage collection [7, 26, 51] or the structural features in object-oriented benchmark suites [7, 41, 51]. While important in other domains, these characteristics have no direct equivalent in shell programs; Koala instead focuses on the shell as glue, and creates an evaluation suite for systems that target environments where components are orchestrated in a higher-level computation.

Earlier work also studies the behavior of programs in simulation or on hardware [23, 28, 38]. Koala's characterization focuses on properties that are independent of any particular hardware or operating system implementation. Koala will make it easy to use shell programs as workloads in the evaluation of general-purpose computational substrates.

Evaluations of shell programs: A variety of systems from several different authors attempt to operate on, optimize, or accelerate shell programs—*e.g.*, achieving elision [4], parallelization [95], fusion [40], synthesis [83], distribution [79], mobile [101], serverless [57], and syscall refinement [27]. These further demonstrate the acute need for a standardized, usable, and replicable benchmark suite for the shell.

9 Conclusion

Benchmark programs are crucial for evaluating ideas, comparing and contrasting approaches, and fueling academic and industrial research. They are especially needed in systems research, where many of the key theses revolve around performance-related arguments and their quantitative evaluation. This need is particularly acute in the context of the shell, where no benchmark suite currently exists.

Acknowledgments

We thank the ATC'25 paper reviewers, artifact reviewers, and several attendees who provided thoughts and feedback. We are especially thankful to Maria Carpen-Amarie, Eric Eide, and Doug McIlroy who offered questions and input at various occasions. We are also grateful to the Brown CS2952R (F'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 Google ML-and-Systems Junior Faculty Award; a Fall'24 Amazon Research Award; a seed grant from Brown University's Data Science Institute; and a BrownCS Faculty Innovation Award.

References

[1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

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

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

[4] Emery D. Berger. Optimizing shell scripting languages. Technical Report UMCS TR-2003-009, University of Massachusetts Amherst, 2003.

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

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pages 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

[8] Enrico Cappellini, Frido Welker, Luca Pandolfi, Jazmín Ramos-Madrigal, Diana Samodova, Patrick L Rüther, Anna K Fotakis, David Lyon, J Víctor Moreno-Mayar, Maia Bukhsianidze, Rosa Rakownikow Jersie-Christensen, Meaghan Mackie, Aurélien Ginolhac, Reid Ferring, Martha Tappen, Eleftheria Palkopoulou, Marc R Dickinson, Thomas W Stafford, Jr, Yvonne L Chan, Anders Götherström, Senthilvel K S S Nathan, Peter D Heintzman, Joshua D Kapp, Irina Kirillova, Yoshan Moodley, Jordi Agusti, Ralf-Dietrich Kahlke, Gocha Kiladze, Bienvenido Martínez-Navarro, Shanlin Liu, Marcela Sandoval Velasco, Mikkel-Holger S Sinding, Christian D Kelstrup, Morten E Allentoft, Ludovic Orlando, Kirsty Penkman, Beth Shapiro, Lorenzo Rook, Love Dalén, M Thomas P Gilbert, Jesper V Olsen, David Lordkipanidze, and Eske Willerslev. Early pleistocene enamel proteome from dmanisi resolves stephanorhinus phylogeny. *Nature*, 574(7776):103–107, October 2019.

[9] Armando Cerna. Pacaur building script. https://github.com/armandocerna/dotfiles/blob/master/scripts/pacaur.sh. Accessed: 2025-01-13.

[10] Andy Chu and Contributors. Oils benchmarks. https://github.com/oils-for-unix/oils/tree/master/benchmarks, 2021. Accessed: 2025-04-28.

[11] Kenneth Ward Church. Unix for poets, 1994.

[12] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: Extending MNIST to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926, 2017.

[13] Cora Coleman, William G. Griswold, and Nick Mitchell. Do cloud developers prefer CLIs or web consoles? CLIs mostly, though it varies by task. *arXiv preprint arXiv:2209.07365*, 2022.

[14] Wikipedia contributors. Wikipedia: Database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download. Accessed: 2025-01-13.

[15] 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.

[16] 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.

[17] Elias Dabbas. Web server access logs. https://www.kaggle.com/datasets/eliasdabbas/web-server-access-logs, 2020. Accessed June 3, 2025.

[18] Martijn Dekker. Modernish. https://github.com/modernish/modernish, 2016. Accessed: 2025-06-02.
[19] Oh My Zsh developers. Oh my Zsh - a delightful and open source framework for Zsh. https://ohmyz.sh/, 2025. Accessed: 2025-12-17.
[20] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. Bash in the wild: Language usage, code smells, and bugs. *ACM Transactions on Software Engineering and Methodology*, 32(1), February 2023.
[21] Adam Drake. Command-line tools can be 235x faster than your hadoop cluster. https://adamdrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html, 2014. Accessed: 2025-06-01.
[22] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium (USENIX Security '13)*, USENIX Security '13, page 605–620, USA, 2013. USENIX Association.
[23] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. *SIGPLAN Not.*, 38(11):169–186, October 2003.
[24] Paul Falstad. *Z shell (zsh)*, 2022.
[25] 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, Renton, WA, July 2019. USENIX Association.
[26] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. The MIT Press, 08 1985.
[27] Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 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.
[28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.
[29] Inc. GitHub. The top programming languages. https://octoverse.github.com/2022/top-programming-languages, 2022.
[30] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the POSIX shell. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.
[31] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. The future of the shell: Unix and beyond. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 240–241, New York, NY, USA, 2021. Association for Computing Machinery.
[32] 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*, HotOS '21, pages 104–111, New York, NY, USA, 2021. Association for Computing Machinery.
[33] S. Greenberg and I.H. Witten. Directing the user interface: How people use command-based computer systems. *IFAC Proceedings Volumes*, 21(5):349–355, 1988. 3rd IFAC Conference on Analysis, Design and Evaluation of Man-Machine Systems 1988, Oulu, Finland, 14-16 June 1988.
[34] The Open Group. The test environment toolkit. https://tetworks.opengroup.org. Accessed: 2025-01-01.
[35] The Open Group. VSCPCTS 2016 test suite. https://www.opengroup.org/testing/testsuites/vscpcts2016.htm. Accessed: 2025-01-01.
[36] The SAM/BAM Format Specification Working Group. Sequence alignment/map format specification. https://samtools.github.io/hts-specs/SAMv1.pdf, November 2024. Version 1.6, last modified on 6 Nov 2024. Accessed: 2025-01-13.
[37] Michael S. Hart and Project Gutenberg. Project gutenberg. https://www.gutenberg.org, 1971.
[38] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 281–296, New York, NY, USA, 2005. Association for Computing Machinery.
[39] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), November 2020.
[40] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. Boosting efficiency of external pipelines by blurring application boundaries. In *Conference on Innovative Data Systems Research (CIDR 2022)*. www.cidrdb.org, 2022.
[41] Urs Hölzle and David Ungar. Do object-oriented languages need special hardware support? In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, page 283–302, Berlin, Heidelberg, 1995. Springer-Verlag.
[42] Fadhl Ibrahim, Julian Oppelt, Manolis Maragkakis, and Zissimos Mourelatos. TERA-Seq: true end-to-end sequencing of native RNA molecules for transcriptome characterization. *Nucleic Acids Research*, 49(20):e115, 2021.

[43] Abebe Israel. Vps audit. https://vpsaudit.vernu.dev. Accessed: 2025-01-13.
[44] Nicolas Jeannerod, Yann Régis-Gianas, and Ralf Treinen. Having fun with 31.521 shell scripts. HAL preprint, April 2017.
[45] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.
[46] 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.
[47] 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.
[48] 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. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 3992–4003, 2023.
[49] Evangelos Lamprou. Foundation models and Unix. *Paged Out!*, 6:9, March 2025.
[50] Evangelos Lamprou, Tianyu (Ezri) Zhu, Di Jin, Grigoris Ntousakis, Georgios Liargkovas, Calvin Eng, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. Controlling opaque-component effects with semisolates and try. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)*. USENIX Association, 2026.
[51] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. A comprehensive Java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering*, ICPE '17, page 3–14, New York, NY, USA, 2017. Association for Computing Machinery.
[52] Georgios Liargkovas, Di Jin, Tianyu (Ezri) Zhu, Dan Liu, A. Bolun Thompson, Anirudh Narsipur, Seong-Heon Jung, Siddhartha Prasad, Diomidis Spinellis, Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. hs: Speculative script reordering at subprocess granularity. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)*. USENIX Association, 2026.
[53] 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*, HotOS '23, page 103–109, New York, NY, USA, 2023. Association for Computing Machinery.
[54] libdash developers. `libdash`. https://github.com/binpash/libdash.
[55] Arch Linux. Arch user repository (AUR). https://aur.archlinux.org. Accessed: 2025-01-13.
[56] Kirk D. Lucas and Contributors. UnixBench: The byte UNIX benchmark suite. https://github.com/kdlucas/byte-unixbench, 2012. Accessed: 2025-04-28.
[57] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. 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.
[58] Marek Majkowski. When bloom filters don't bloom. https://blog.cloudflare.com/when-bloom-filters-dont-bloom, March 2 2020. Accessed: 2025-01-13.
[59] M Douglas McIlroy. Coroutine prime number sieve, 2014.
[60] 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.
[61] Koichi Nakashima. ShellBench: A POSIX shell benchmark suite. https://github.com/shellspec/shellbench, 2021. Accessed: 2025-04-28.
[62] Koichi Nakashima and contributors. ShellSpec - a full-featured bdd framework for shell scripts. https://shellspec.info. Accessed: 2025-01-01.
[63] 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.
[64] Netresec. Publicly available pcap files. https://www.netresec.com/?page=PcapFiles, 2025. Accessed: 2025-06-02.
[65] Inc. npm. npm.js. https://www.npmjs.com. Accessed: 2025-01-13.
[66] National Oceanic and Atmospheric Administration. National oceanic and atmospheric administration (NOAA). https://www.noaa.gov. Accessed: 2025-01-13.
[67] Brown University Department of Computer Science. CSCI 1380: Distributed computer systems. https://cs.brown.edu/courses/csci1380, 2025. Accessed: 2025-06-04.
[68] Ollama. Ollama: Run large language models locally. https://ollama.com, 2023. Accessed: 2025-06-02.
[69] OpenAI. *OpenAI API: Embeddings Guide*, 2024. Accessed: 2025-06-02.

[70] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.

[71] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[72] Roman Perepelitsa and Contributors. zsh-bench: Benchmark for interactive Zsh. https://github.com/romkatv/zsh-bench, 2021. Accessed: 2025-04-28.

[73] Stéphane Peter. makeself: Make self-extractable archives on Unix. https://makeself.io. Accessed: 2025-01-13.

[74] Alina Petukhova, Jo ao P. Matos-Carvalho, and Nuno Fachada. Text clustering with large language model embeddings. *International Journal of Cognitive Computing in Engineering*, 6:100–108, 2025.

[75] Meikel Poess and Chris Floyd. New TPC benchmarks for decision support and web commerce. *SIGMOD Rec.*, 29(4):64–71, December 2000.

[76] Bats-Core Project. Bats-Core: Bash automated testing system. https://bats-core.readthedocs.io/en/stable. Accessed: 2025-01-01.

[77] The Nmap Project. Nmap: Network mapper. https://nmap.org/, 2025. Accessed: 2025-12-17.

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

[79] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: a data-aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*, USENIX ATC '20, USA, 2020. USENIX Association.

[80] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *CACM*, 17(7):365–375, July 1974.

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

[82] U.S. Securities and Exchange Commission. Edgar log file data sets. https://www.sec.gov/data-research/sec-markets-data/edgar-log-file-data-sets, 2024. Accessed June 3, 2025.

[83] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. 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.

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

[85] Piotr Sykulski. shnake: Simple snake game in Bash. https://github.com/psykulsk/shnake, 2025. Accessed: 2025-12-17.

[86] Ole Tange. GNU parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[87] Dave Taylor and Brandon Perry. *Wicked Cool Shell Scripts: 101 Scripts for Linux, OS X, and UNIX Systems.* No Starch Press, 2016.

[88] Gemma Team. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[89] The Netfilter Core Team. iptables: Userspace packet filtering firewall. https://www.netfilter.org/projects/iptables/, 2025. Accessed: 2025-12-17.

[90] Eleftheria Tsaliki and Diomedes Spinellis. The real numbers for athens buses. https://insidestory.gr/article/noymera-leoforeia-athinas, 2020.

[91] Edward Tufte. New york city weather chart. https://www.edwardtufte.com/notebook/new-york-city-weather-chart, 2004. Accessed: 2025-06-02.

[92] Justine Tunney. Bash one-liners for LLMs. https://justine.lol/oneliners, 2023. Accessed: 2025-06-01.

[93] Office of the Chief Actuary U.S. Social Security Administration. Popular baby names data. https://www.ssa.gov/oact/babynames/limits.html, 2025. Accessed: 2025-12-17.

[94] Saiteja Utpala, Alex Gu, and Pin-Yu Chen. Language agnostic code embeddings. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 678–691, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[95] 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*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[96] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node via RWX-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1838, 2021.

[97] Valdemar Švábenský, Jan Vykopal, Pavel Seda, and Pavel Čeleda. Dataset of shell commands used by participants of hands-on cybersecurity training. *Data in Brief*, 38:107398, 2021.

[98] Kate Ward. shUnit2 - xunit unit testing framework for bourne based shell scripts. https://github.com/kward/shunit2. Accessed: 2025-01-01.

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

[100] Simon Willison. LLM: A cli tool and python library for interacting with large language models. https://llm.datasette.io. Accessed: 2025-06-02.

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

[102] Yizheng Xie, Evangelos Lamprou, Jerry Xia, and Nikos Vasilakis. Incr: Faster re-execution via bolt-on incrementalization. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)*. USENIX Association, 2026.

[103] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: standardizing and benchmarking interactive coding with execution feedback. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems*, NeurIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[104] Tianqi Zhao, Ming Kong, Tian Liang, Qiang Zhu, Kun Kuang, and Fei Wu. CLAP: Contrastive language-audio pre-training model for multi-modal sentiment analysis. In *Proceedings of the 2023 ACM International Conference on Multimedia Retrieval*, ICMR '23, page 622–626, New York, NY, USA, 2023. Association for Computing Machinery.