bench: A Programmable Command-Line Framework for Complex Benchmarking Workflows

Filip Říha
filip.riha@fit.cvut.cz
Czech Technical University
Prague, Czech Republic

Filip Křikava
filip.krikava@fit.cvut.cz
Czech Technical University
Prague, Czech Republic

Abstract

Benchmarking provides much of the feedback that drives systems and language research, yet sound benchmarking is hard. As experiments grow, added programs, configurations, custom metrics, warmup, failures, and machine noise can all mislead. We present bench, a tool that keeps simple measurements simple and lets complex experiments scale without leaving a general-purpose language. It offers a command-line interface for ad-hoc runs and a small, almost declarative Python API for repeatable experiments, both backed by one engine. A builder API resolves a suite of benchmarks lazily against a typed context, so a single script can be parametrized from the command line, discover its workloads at run time, and treat process benchmarks and in-process harnesses through one pipeline. We have used it to simplify the benchmarking infrastructure for our R just-in-time compilers, to evaluate student language implementations in a course, and to drive established suites such as Renaissance and SPEC.

CCS Concepts: • **Software and its engineering → Software performance**; *Empirical software validation*; *Software testing and debugging*.

Keywords: benchmarking,testing,tool,python

ACM Reference Format:
Filip Říha and Filip Křikava. 2026. bench: A Programmable Command-Line Framework for Complex Benchmarking Workflows. In *Companion Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3837729.3840502

1 Introduction

Benchmarking is a basic feedback loop for anyone who optimizes software: we want to know whether a change makes a system faster, and by how much. Getting a trustworthy answer, however, is notoriously difficult. Good representative workloads are hard to find, a reproducible setup is hard to build, and the resulting numbers are hard to interpret. Measurement noise, non-determinism, just-in-time warmup, and a long list of environmental factors can easily produce wrong conclusions from data that looks clean [1, 5, 11].

The tools that support this loop tend to sit at one of two ends. At one end are lightweight command-line tools that make the common case effortless. `hyperfine` [13] is a fine example: it measures wall-clock time, computes summary statistics, detects outliers, and compares a baseline against a new version, all behind a friendly interface. This ease of use matters, because without it people simply do not benchmark, or benchmark badly. But such tools are deliberately narrow. `hyperfine` handles a single benchmark at a time, measures wall-clock time and little else, cannot capture a custom metric such as throughput, and has no model for a harness that iterates a workload inside one long-running process, making it unsuitable for a large class of benchmarks.

At the other end are heavyweight, declarative frameworks. ReBench [9, 10] is a powerful and mature example, well suited to long-lived setups such as tracking a virtual machine's performance over time. Its experiments live in a single YAML configuration, which is ideal for a fixed, recurring study but rigid for exploratory work: an experiment cannot easily be parametrized from outside its configuration, so even modest needs such as supplying several input or output paths become cumbersome, and the fixed benchmark structure is awkward to extend. Standing one up is also a heavier commitment than many questions warrant.

Most real benchmarking falls between these two ends, or beyond both. The moment an experiment outgrows the simple tool yet does not fit the declarative one, the usual recourse is a shell script that wires programs together and reinvents argument parsing, repetition, statistics, and orchestration. The typical trajectory is incremental: one times a single command, wraps it in a loop for a few samples, then adds warmup, a timeout, and summary statistics, and before long has hand-rolled a small and fragile benchmarking framework sprinkled with undocumented environment variables. This is a great deal of wheel reinventing for problems the community has solved many times over.

Our starting point is a simple observation: a benchmark driver is, at its core, just a program. It runs a workload some

SPLASH Companion '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2910-2/2026/10
https://doi.org/10.1145/3837729.3840502

number of times, reads numbers back out, and summarizes them. If we treat it as exactly that, a general-purpose language equipped with a small, well-chosen set of benchmarking operations, then a simple measurement can stay a one-liner, while anything more elaborate is just more code, with neither a rigid configuration to outgrow nor a cliff that forces a drop to a shell script.

We present bench, a tool built on this idea: easy to use for simple measurements, yet able to scale to complex benchmarking scenarios. It is a command-line tool in the spirit of `hyperfine` and a Python library for the workflows that outgrow simple commands. The two are not separate tools: the command line is a thin layer built using the same user-facing API. In summary, bench provides

- ad-hoc command-line benchmarking with built-in warmup, repetition, summary statistics, and baseline comparison,
- static and dynamic, context-dependent configuration,
- a cross-product matrix of variants, with per-cell filtering,
- harness benchmarks that iterate inside one process,
- custom metrics with built-in support for CPU time, resource usage, or hardware counters,
- composable stopping policies,
- outlier detection and failure handling,
- exporting results to JSON, CSV, or a per-run directory,
- listing, filtering, dry-run, and parallel execution,
- environment checks and machine-noise reduction on Linux.

2 Using bench

bench has two faces backed by one engine. This section works through them from the outside in: the command line for ad-hoc measurements, the Python API for experiments that outgrow a single command, and the support for trustworthy results. Section 3 then turns to the model that underlies all of it.

2.1 The Command-Line Interface

For ad-hoc work bench behaves similarly to `hyperfine`. The `bench run` command benchmarks one or more programs with warmup, repetition, and statistics. Each command gets its own summary statistics. All variants of a single benchmark are then compared against one another, providing immediate feedback. There are flags to persist the results and builtin comparison.

`bench run` is not a separate engine. It parses its arguments, builds exactly the `suite(...)` a user would write by hand. An ad-hoc matrix is even available with `-M`, but from the command line bench cannot know that two cells are different programs rather than variants of one benchmark. For example we cannot easily compare two CPython versions on a multiple benchmarks. Expressing that is where the Python API takes over.

2.2 The Python API

The API is almost declarative, but it is plain Python, so the full power of a general-purpose language is always at hand. The following script compares two Python interpreters across three workloads, measuring both wall-clock time and peak memory. The command is a function of the context, reading the chosen interpreter and the benchmark name, so each of the six variants resolves its own command. Unlike the command-line matrix, variants are ranked within each benchmark and never across. From a single run bench reports summary statistics for every variant (mean, median, standard deviation, and range), ranks and compares the variants of each benchmark, and summarizes each suite with an overall geometric mean across its benchmarks. We now refine this script to demonstrate some of the essential features.

```
s = (
    suite("interpreters")
    .add(bench("fib")).add(bench("nbody"))
    .add(bench("json"))
    .with_matrix(vm=["python3.12", "python3.13"])
    .with_command(lambda ctx: [
         ctx.matrix.vm, f"{ctx.benchmark}.py"])
    .with_process_metric(Time(), max_rss())
    .with_warmup(2).with_runs(10)
)
run(s) # execute the bench driver
```

Command-line parameters. A script declares its flags as a Python dataclass[1], and `run(s, params=Params)` builds a complete command-line interface from the field annotations, exposing the typed instance as `ctx.params` and providing common flags (*e.g.*, `--list`, `--dry`, `--jobs`, `--include`) for free.

```
@dataclass
class Params:
    runs: int = 10
s = s.with_runs(lambda ctx:
    FixedRuns(ctx.params.runs))
```

Each field of `Params` becomes a command-line flag, and its value is available through `ctx.params` in any function-valued field.

Discovering benchmarks. Instead of listing benchmarks by hand, a factory produces them at materialize time, so it can inspect the filesystem or the parameters. Here it discovers every benchmark program in a folder, still crossed with the interpreter matrix.

```
s.factory(lambda ctx:
    [bench(p.stem, path=p) for p in
    Path(ctx.params.base_dir).glob("*.py")])
```

Custom metrics. By default bench measures wall-clock time, but any metric can be configured. There are two types

[1] PEP 557 – Data Classes, *cf.* https://peps.python.org/pep-0557/

of metrics: an iteration metric or a process metric. The difference is when do they run and from which source they extract data. The iteration metrics run on each iteration, parsing the output of an iteration (often a block of text from standard output). The process metrics run at the command exit and have access to all of its properties (*e.g.*, output, rusage struct). We provide a few built-in metrics: `Time` measures wall-clock (and user/system time), resource usage metrics such as `max_rss`, and hardware counters on Linux through `PerfStat` (*e.g.*, cache misses, branch misses).

Harness benchmarks. Some workloads must iterate inside one process. A just-in-time compiler such as PyPy, for instance, needs many iterations before it reaches a steady state [1]. `with_harness()` marks a benchmark as such: the command runs once, and each iteration it produces becomes one run, the leading warmup ones dropped from the statistics. When an iteration spans several lines, a monitor frames the output stream, which is how real suites such as Renaissance and LevelDB's `db_bench` are supported.

2.3 Trustworthy Measurement

Two commands address the noise that undermines results. `bench doctor` inspects the machine for conditions that add variance, prints a snapshot of the environment with a list of warnings. `bench denoise` goes further and quiets the machine (currently Linux only). It sets every CPU to the performance governor, disables Turbo Boost, allows performance-counter access, and turns off swapping, address-space randomization, and transparent huge pages, saving the originals so it can restore them afterward or after a crash. The same can be folded into a run with `bench run --denoise`.

Where those commands tackle the machine, bench also checks the measurements themselves. Each metric's samples are screened for outliers with the modified Z-score test [7]. Any detected outlier is marked in the reports and a warning message is displayed.

3 The bench Model

Having seen bench in use, we now describe the model underneath. bench separates the description of an experiment from its execution: the user constructs an immutable description through a builder API, and a runtime resolves and executes it.

3.1 The Benchmark Definition Model

The interpreter script of Section 2 is a small instance of the model. A `SuiteBuilder` is a named collection of benchmarks together with the defaults they share, and a `BenchmarkBuilder` is the specification of one workload, its command, working directory, environment, metrics, and stopping policies. In that script, `suite("interpreters")` is the `SuiteBuilder`, each `bench(...)` adds a benchmark builder. The trailing `with_*` calls set the defaults the three benchmarks share. Two ideas make this specification flexible. First, every field is either a static value or a function of a `Context`, resolved lazily later. For example, the `with_command` call computes each variant's command from the context. The context carries the typed user parameters parsed from the command line, the suite and benchmark names, and the values of the current matrix cell, so a command, a timeout, or an environment variable can be computed from any of them. Second, a benchmark may declare a matrix, a set of named dimensions whose cartesian product becomes its variants: `with_matrix(vm=[...])` crosses two interpreters with the three benchmarks to give the six variants. A function-valued field reads the current cell through the context, and a dimension's values may themselves be computed from the context.

Configuration is resolved lazily and in one place. A field left unset inherits the suite default, factories run to produce further benchmarks (the discovery script of Section 2 globs the filesystem this way), and the matrix is expanded, all at materialize time. Because resolution is deferred to this single pass, the order of builder calls never matters and a per-benchmark value always wins over the suite default. Each surviving cell is then resolved into one immutable `Benchmark` carrying fixed execution instructions ready to run. This is why the summary of the six-variant script ranks variants within each benchmark and never across: different programs are not directly comparable.

3.2 Execution Pipeline

Figure 1 shows how resolved benchmarks are run. `plan` flattens all suites into one list of `Benchmark` variants, which `select` narrows by include and exclude regular expressions. A `Runner` then drives the list. `Sequential` runs one benchmark at a time (the only sound choice when measuring performance), while `Parallel` runs several at once (useful for testing and debugging), and `Dry` only enumerates what would run.

Each benchmark is driven by a `Controller`, a small feedback loop that pulls one `Iteration` at a time and feeds it to a `StoppingPolicy` until enough have been collected. The policy is split into a frozen configuration and a mutable per-run observer, which keeps the description immutable while each run gets its own live state, and the leading warmup iterations are dropped from the statistics but still recorded.

Where iterations come from is the responsibility of a `RunSource`, and this is what unifies the two kinds of benchmark. The default `CommandSource` spawns one process per iteration, whereas `with_harness()` selects a `HarnessSource` that spawns a single long-running process whose output a monitor frames into many iterations, all belonging to one run. The latter matters for warmup-sensitive workloads such as a just-in-time compiler, where iterations must share one process so the machine stays warm. Either way an execution yields an `ExecutionResult`, a success predicate turns it into a verdict, and a failed run still produces a `Run`

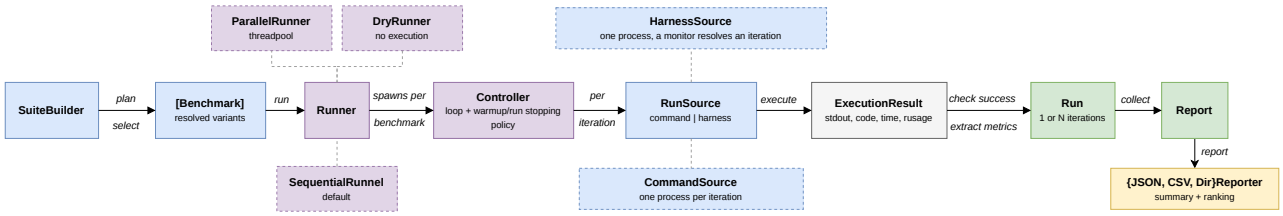

Figure 1. Execution pipeline

with its output preserved rather than fake numbers. Metrics come in two kinds that mirror this split. An `IterationMetric` parses the text of one iteration, as `FloatPerLine` does, while a `ProcessMetric` reads the whole process once, such as the `max_rss()` of the interpreter script, which reports peak memory from `getrusage`. The resulting `Run` s accumulate in a `Report` that round-trips through JSON, while streaming reporters emit progress, JSON, CSV, or a per-execution directory tree and a formatter prints the final summary.

4 Evaluation

We have exercised bench in three settings. First, we ported the benchmarking infrastructure of the R just-in-time compilers we develop [4, 8]. One ReBench-based setup, previously 120 lines of YAML together with three Bash scripts totaling 140 lines, became a single 120-line Python script. Those Bash scripts had existed only to rewrite the ReBench configuration, adjusting benchmark selection and parameters to the running environment. bench absorbs that logic directly and turns a collection of undocumented environment variables into command-line parameters. A second, ad-hoc setup of 150 lines of Bash and a 10-line Makefile collapsed to a 45-line Python script, and that figure does not even count the separate scripts that computed statistics. bench does not itself produce plots or elaborate statistical reports, yet its immediate console feedback has proved valuable in practice: because a measurement is now cheap to express, we benchmark more readily instead of waiting on a long pipeline.

Second, we use bench in a runtime-systems course in which students build a bytecode interpreter for the Lox language and must carry out a proper evaluation. We adopted it not only for benchmarking but also for testing and building the submissions. For testing, a suite holds one benchmark per test, run once and in parallel, with a custom success predicate that compares a submission's output against the expected output. For building, we effectively benchmark each submission's build, using custom metrics to extract the build time, the source lines of code, and the binary size. The decisive advantage is that bench orchestrates the commands and records their standard output, standard error, and exit code, so the results can be analyzed afterwards. This replaced the course's Makefiles and shell scripts, reducing them on average by a factor of two.

Finally, we drive several established suites, namely Renaissance, SPEC, CPython, and LevelDB. These exercise different parts of the tool: SPEC reports to a log file rather than to standard output, and LevelDB reports throughput rather than running time. Here the principal benefit is a tailored benchmarking command-line tool: it presents the comparison in a usable console interface, emits results in a predictable format that includes a snapshot of the running environment, and collects errors as it proceeds.

5 Related Work

bench sits between `hyperfine` [13] and the more configuration-driven tools. ReBench [9, 10] is the closest in spirit and the most mature, but, as discussed, its single YAML configuration makes exploratory, parametrized work cumbersome, the niche bench targets with a programmable API. benchkit [12], a recent declarative Python framework, shares our goal of replacing ad-hoc shell scripts but emphasizes composing existing system tools through campaigns, wrappers, and hooks while keeping benchmark code untouched, whereas bench starts from the immediacy of a command line and grows into a programmable Python API with a uniform process and harness pipeline. temci [2] pioneered practical machine-quieting, which inspired our `denoise` command. Our design is further informed by the methodology literature on trustworthy results [5, 11], warmup [1], statistically sound evaluation [3], and outlier handling [6, 7].

6 Conclusion

We have presented bench, a benchmarking framework built on the premise that a benchmark driver is, at its core, just a program. By pairing a small set of well-chosen benchmarking operations with a general-purpose language, bench keeps simple measurements simple while letting complex experiments scale, and it leaves the full power of Python available for everything in between. A command-line interface and an almost-declarative Python API share a single engine, so a study can grow from a one-line comparison into a parametrized, multi-dimensional suite without ever falling back on a bespoke shell script.

Use of AI

While working on the tool, we used AI to help with documentation, streamlining the test suite, migrating examples across breaking API changes, and code reviews.

Tool Availability

The tool is available as an open-source project under the MIT license at https://github.com/PRL-PRG/bench. A short demo can be seen at https://youtu.be/0D-BJz6ayNQ. A snapshot of the version used for this submission is archived at https://doi.org/10.5281/zenodo.20960758.

Acknowledgements

This work was supported by the Czech Science Foundation grant 23-07580X.

References

[1] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold, In Proceedings of the ACM on Programming Languages (OOPSLA). *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 52, 27 pages. doi:10.1145/3133876

[2] Johannes Bechberger. [n. d.]. temci: An Advanced Benchmarking Tool. https://github.com/parttimenerd/temci.

[3] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 219–228. doi:10.1145/2451116.2451141

[4] Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *PACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428288

[5] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 57–76. doi:10.1145/1297027.1297033

[6] Google. [n. d.]. Reducing Variance — Google Benchmark User Guide. https://google.github.io/benchmark/reducing_variance.html.

[7] Boris Iglewicz and David C. Hoaglin. 1993. *How to Detect and Handle Outliers*. The ASQC Basic References in Quality Control: Statistical Techniques, Vol. 16. ASQC Quality Press, Milwaukee, WI.

[8] Matěj Kocourek, Filip Křikava, and Jan Vitek. 2025. Copy-and-Patch Just-in-Time Compiler for R. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '25)*. doi:10.1145/3759548.3763370

[9] Stefan Marr. 2018. ReBench: Execute and Document Benchmarks Reproducibly. doi:10.5281/zenodo.1311762 Software, https://github.com/smarr/rebench.

[10] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking: Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS)*. 120–131. doi:10.1145/2989225.2989232

[11] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 265–276. doi:10.1145/1508244.1508275

[12] Antonio Paolillo, Mats Van Molle, and Ken Hasselmann. 2026. sbenchkit: A Declarative Framework for Composable Performance Evaluation of System Software. In *Proceeding of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE)*. doi:10.1145/3777884.3796997

[13] David Peter. [n. d.]. hyperfine: A Command-Line Benchmarking Tool. https://github.com/sharkdp/hyperfine.

Received 2026-06-27; accepted 2026-07-25