Dear Professor Templ and Reviewers,

Thank you for the constructive reviews of our manuscript and for the opportunity to submit a minor revision. We have rerun the benchmarks with `rush` 1.3.0 and added five replications to the Bayesian optimization benchmark. We have also expanded the descriptions of the software environment, testing, and concurrency mechanisms, improved the figures, and added a competing-interests declaration. Section, table, and figure references refer to the revised manuscript.

Editor

1. [software-quality] Version inconsistency in the archived artifact. The Zenodo record (https://zenodo.org/records/22256435) is titled "benchmark 2026 rush 1.1.1", while Sections 4.1 and 5.1 state the benchmarks use rush 1.2.0. Reconcile the artifact label with the version actually used, or note which package version the deposited code pins, so the benchmarks can be reproduced against the correct release.

Version 1.1.1 referred to the benchmark repository, not the `rush` package. We reran the benchmarks with `rush` 1.3.0. The record name "benchmark 2026 rush 1.3.0" identifies the package version used now (https://zenodo.org/records/22899537).

2. [statistical-rigor] The headline BO comparison in Section 5 rests on n=1: "Each task–algorithm combination is run once." Table 2 reports utilization (94.81%, 95.62%, etc.) and evaluation counts to several significant figures, and the text claims "roughly 300 times higher utilization", with no replication and no dispersion measure. The reader cannot tell how much of the gap is stable versus run-to-run scheduling variance. The effects span orders of magnitude, so the qualitative conclusion is plausible, but the precision implied by the reported digits is unsupported. Add at least two or three replicate runs per task–algorithm cell and report a range or standard deviation, or qualify the Section 5 numbers explicitly as a single-run illustration and reduce the reported precision (for example " 95%","roughly two orders of magnitude") rather than "94.81%".

We repeated the Section 5 benchmarks with `rush` 1.3.0 and five replications per task-algorithm combination. Table 2 and 6 now report means and standard deviations. Figure 5 shows the mean incumbents and the shaded bands representing the best and worst replications. For each task, ADBO utilization varies by less than one percentage point across replications. This variation is small relative to the differences between algorithms. On credit-g, ADBO utilization is 270 to 300 times that of CL and ACBO. The introduction and conclusion now report utilization above 93% in every ADBO run. The largest variation occurs on airlines. Here, ACBO utilization ranges from 20% to 41%, and ADBO utilization is 2.4 to 4.9 times that of ACBO in the corresponding replications. Final classification errors remain similar across methods. ADBO has the lowest mean error on all four tasks. The differences are small: up to 0.8 percentage points on credit-g and less than 0.1 percentage points on the other tasks. The discussion in section 6 states that the benchmark is not designed to assess final optimization performance.

3. [consistency-check] Figure 5's caption states the algorithms "start from the same initial design of 100 configurations that was evaluated before the timed runs (Section 5.1)", but Section 5.1 does not mention a 100-point initial design (the only initial designs described are the 25-point toy example in Section 3.2). Add the initial-design description to Section 5.1 or correct the cross-reference, since the figure points readers to text that does not contain the stated fact.

Section 5.1 now describes the initial design. We randomly sampled initial designs from the search space and evaluated them using a 10-fold cross-validation. We did this for each replication and task to ensure a consistent starting point across the algorithms.

4. [software-quality] The paper does not describe a test suite. For a coordination layer whose value rests on concurrency, error handling, and lost-worker detection, the absence of any stated tests for these paths is the main gap. State whether the package ships unit tests that exercise the documented failure modes (worker crash, heartbeat expiry, failed tasks) and the caching invariants, so a reader can judge coverage beyond the happy path.

The new Section 2.9 describes the test suite. `rush` includes 105 unit tests implemented with `testthat` that run on GitHub Actions on the release and development versions of R. The tests cover the core functionality, error handling, lost-worker detection, concurrent access, and caching behavior of the package.

5. [software-quality] Portability depends on an external Redis server that must be installed and started separately, and the multi-node runs use a specific LRZ cluster. The install link is provided and mirai handles remote/SSH/SLURM deployment, so this is a declared caveat, not a hidden one. Record the declared dependency versions (redux, mirai, ranger, lhs, data.table) alongside the Redis version used, so a new user can build the same environment.

Section 4.1 now lists the versions of R, Redis, and `rush`. The benchmark repository includes `setup_hpc.sh` to install these versions. The `renv.lock` file records the whole R package environment including ranger, lhs and data.table.

6. [software-quality] CRAN/pkgcheck readiness cannot be judged from the text. The paper does not state whether rush itself is on CRAN or passes R CMD check, though several dependencies (bbotk, mlr3tuning) are cited as released packages. State the package's distribution channel and check status.

Section 2 now states that `rush` is available on CRAN, with version 1.3.0 current at the time of revision. The development version is hosted on GitHub. Having the package available on CRAN means that `R CMD check` passes.

7. [statistical-rigor] Comparative claims about budget adherence ("CL can overshoot … sometimes substantially") are drawn from single wall-clock values in Table 6 with no uncertainty, so the direction is asserted rather than demonstrated across runs. Tie this claim to the same replicate runs suggested above, or state that it is a single-run observation.

The budget comparison now uses five replications per task-algorithm combination. Table 6 reports the mean and standard deviation of wall-clock time.

8. [visualisation] Figure 3 plots runtimes that span from about 1 ms to over 40,000 ms (Table 3). State explicitly in the caption whether the y axis is log-scaled and label its units, since a linear axis would compress the cached curve to a flat line and hide the very speedups the figure claims.

The y axis now uses a logarithmic scale and includes the unit in its label. The caption states that both axes are log-scaled.

9. [visualisation] Figures 3 and 4 encode the two conditions as red and blue. Red versus blue is generally distinguishable under the common red-green deficiencies, so this is acceptable, but

add a redundant channel (line type in Figure 3, fill pattern or direct labels in Figure 4) so the distinction survives greyscale printing.

Figure 3 now uses circles for cached fetching and triangles for fetching without caching with an explantion in the caption. In Figure 4, bar height already distinguishes the phases: proposal uses one core, and evaluation uses ten. We also use light and dark fills to distinguish the phases in grayscale. The caption describes these fills.

10. [visualisation] Figure 5 overlays three algorithm traces (CL, ACBO, ADBO). Confirm the three lines use a colourblind-safe palette plus distinct line types or direct end-of-line labels, because three overlapping time series separated by colour alone are the most common readability failure.

Figure 5 now uses orange, blue, and green from the Okabe-Ito palette. The algorithms also have distinct line types: dotted for CL, dashed for ACBO, and solid for ADBO. The caption identifies both encodings. Figures 3 and 4 use the same palette.

11. [ethics-and-integrity] No competing-interest statement appears, though the authors develop the package and related mlr3 ecosystem tools they benchmark. This is standard for open-source method papers and is not an undisclosed financial interest. Add a short competing-interests declaration noting authorship of the rush, bbotk, and mlr3tuning packages.

A competing-interests section now follows the acknowledgments. It states that the authors develop and maintain `rush`, `bbotk`, `mlr3tuning`, and `mlr3mbo`. It also declares that the authors have no financial competing interests.

12. [methods-soundness] The utilization metric U = T optimization/T CPU counts surrogate fitting and acquisition optimization as productive time, and in ADBO each of the 448 workers refits its own surrogate on the full archive. Table 6 shows this dominates: on credit-g the cumulative surrogate time is 212,896 s versus 10,625 s of learner training, so most of the reported 94.81% "utilization" is redundant surrogate computation rather than objective evaluation. The headline "94–99% CPU utilization" therefore measures how busy the cores are, not how much useful optimization work is done. State plainly in Section 5 that a large share of ADBO's counted time is surrogate refitting, and separate the evaluation share from the surrogate/acquisition share when reporting utilization.

We agree that showing the share of time spent on learner training is useful. We added a learner utilization column to table 6 and describe the learner utilization in section 5.1 and 5.2.

However, we believe that surrogate fitting is not "redundant" in the context of ADBO. All three strategies fit one surrogate per proposed configuration and this is part of the optimization process rather than overhead. We think this is likely a (partial) misunderstanding, but also a subtle point, so let us explain this here carefully. Let's start with the metric U. We measure the time where CPUs are active (surrogate fitting, acquisition optimization, and learner training, summed over workers) divided by the complete time of an optim trace. Is this is $< 100\%$ CPUs must have been idle. And compute resources have been "underused", which is bad. We think nobody is disagreeing here. What we take from the editors criticism (repeated by multiple reviewers) is: If this is close to 100% this is not necessarily "ideal" *if* computation is redundantly / unnecessarily replicated i.e. the "redundant" fitting of the surrogates on the archive. But the whole point of ADBO is that this surrogate fitting / acquisition function optimization is *not* redundant as it does not operate on the same state! Let's go through a few examples. Let's take 2 workers. Worker 1 reads the archive. It fits a surrogate and then proposes a point. This point is evaluated. We wait. While we wait

worker 2 reads the archive. This now includes the proposed point of W1 – with an imputed fitness values. State is different. Now it could happen that W2 reads the archive while W1 still proposes. Even then the state or proposal differs: a) The acquisition function is randomized (random lambda parameter in LCB per worker) and there is randomness in the random forest and acquisition function optimizer. b) Even the training data of the surrogate model is most of the time different. We can – in a multi-worker setup – likely assume that SOME points are under eval currently. These are also then imputed randomly. We added a footnote to section 5.1 explaining why surrogate fitting in ADBO is not redundant.

We checked how often the workers fitted a surrogate on the same archive with no running points. In this case the workers would indeed see the same archive because nothing is imputed.
In the first iteration of each worker, this happens often because the workers see only the initial design. But each time a different configuration was proposed due to the randomness in the acquisition function. After the first iteration, we found one case in the 20 ADBO runs where four workers saw the same archive with no running points (First replication of ADBO on credit-g). So redundant work is unlikely to happen during the optimization process.

13. [statistical-rigor] The microbenchmark tables (Table 1, Table 3) report only the median over 10,000 replicates. No spread is given, so the reader cannot judge tail latency, which is the quantity that matters for per-task overhead under contention. The single large-payload cell (17.77 ms) suggests the distribution is heavy-tailed and a median hides this. Add an interquartile range or a high quantile (for example median and 95th percentile) to Tables 1 and 3.

Tables 1 and 3 now report the median with the 95th percentile in parentheses. The largest configuration in Table 1 has 100 fields with 10,000 values each. Its median push time is 12.98 ms, and its 95th percentile is 25.41 ms. The corresponding finish times are 12.07 ms and 20.80 ms. The 95th percentiles are approximately 2.0 and 1.7 times the medians.

Reviewer 1

1. [required] In the abstract and Section 5.2, qualify the 94 to 99 percent utilization result: state that U counts surrogate fitting and model training as productive time, that ADBO's high utilization reflects large and growing surrogate-fitting cost (cite the Table 6 credit-g figures), and that higher utilization did not produce lower final error.

We agree that showing the share of time spent on learner training is useful. We added a learner utilization column to table 6 and describe the learner utilization in section 5.2. Section 5.2 also mentions now that the higher utilization did not translate into lower final error but rather into finding the optimal configurations earlier.

2. [required] State plainly that the Section 4 overhead numbers are single-worker measurements against a local Redis over a Unix socket, and qualify the 448-worker scalability claims accordingly, or add a concurrent-access measurement.

Section 4.1 now specifies that the measurements use a single worker and a local Unix socket. We write in Section 6 now that per-operation latency under concurrent TCP access was not measured separately.

3. [required] Reconcile the rush version between the text (1.2.0) and the Zenodo reference (1.1.1), and pin the exact version and commit used for the benchmarks.

Version 1.1.1 referred to the benchmark repository, not the `rush` package. We reran the benchmarks with `rush` 1.3.0. The record name "benchmark 2026 rush 1.3.0" identifies the package version used now (https://zenodo.org/records/22899537).

4. [recommended] Note that CL evaluates each configuration on 10 cores while ADBO uses one core per evaluation, and explain how this granularity difference affects the evaluation-count comparison.

Section 5.1 now describes how the cores are allocated to each configuration. CL distributes the 10 resampling iterations across 10 cores and evaluates 44 configurations in parallel. ACBO and ADBO run these iterations sequentially on one core, allowing up to 448 concurrent configurations. Each evaluation therefore represents a complete 10-fold cross-validation under all three strategies.

5. [recommended] Add a short paragraph on the shared Redis store as a contention point, single point of failure, and source of memory growth at scale.

The shared Redis server is a single point of failure: coordination stops while it is unavailable. Worker or manager failures, however, do not erase the task state held in Redis. Redis can also be configured to save data to disk.

As more workers access Redis and more results are stored, contention and memory use can increase. In our benchmarks, the largest archive contains approximately 16,000 tasks and occupies a few tens of megabytes; the 448 workers complete at most 27 evaluations per second, each requiring a small number of database commands.

We have not added the suggested paragraph because our benchmarks do not show that contention or memory capacity limits performance at the tested scale. Larger benchmarks would be needed to determine the scale at which Redis contention or memory capacity could impact performance. Redis allows up to 10,000 simultaneous connections in its default configuration.

6. [recommended] Add an explicit single-run caveat to the evaluation-count and final-error results in Table 2 and Table 6.

We reran the benchmark with five replications per task-algorithm combination. Tables 2 and 6 now report means and standard deviations. These summaries show the variation in evaluation counts and final errors across runs.

7. [optional] An asynchronous random-search baseline on the same tasks would isolate how much of ADBO's utilization advantage is specific to Bayesian optimization rather than to asynchronous scheduling alone.

We have not added this now (we are in general not aginst adding more methods), because we politely do not see what this would achieve (in the context of this paper).

a) The paper is about rush, as a computational async infrastructure, which might be especially needed / convenient in the context of HPO.

b) It is not about a general comparison of sync / async optimizers and their general pros and cons. This is why we mainly focus on the technical aspect of "cpu utilization" (and less on "classif error" / quality which we mainly report for completeness sake).

c) Now, the reviewer would like cpu util be reported for async RS, but … that would be ~ 100% by construction? You can argue that a (scientific) comparison of this method vs a sync BO technique (in the case of heterogenous runtimes) is interesting, but this is exactly what we

explained in b) is not the focus of this paper and somewhat out of scope? In our opinion this would also require a much larger / complex benchmark – if done properly.

Reviewer 2

1. [required] Using the Table 6 decomposition, report how much of ADBO's counted CPU time is objective evaluation versus surrogate refitting, or qualify the sentences stating that higher utilization translates directly into more configurations and better performance, so utilization is not read as optimization efficiency.

We agree that showing the share of time spent on learner training is useful. We added a learner utilization column to table 6 and describe the learner utilization in section 5.2. Section 5.2 also mentions now that the higher utilization did not translate into lower final error but rather into finding the optimal configurations earlier.

2. [required] State next to the comparative results that each task-algorithm combination is run once, and limit the evaluation-count and final-error comparisons to qualitative statements; keep the utilization contrast, which is large enough to hold.

We reran the benchmark with five replications per task-algorithm combination. Sections 5.1 and 5.2 state the replication count. Tables 2 and 6 report means and standard deviations for utilization, evaluation counts, and final classification errors. The utilization advantage persists across all replications. Sections 5.2 and 6 report similar final errors and describe the small observed differences. They do not claim a general advantage in final optimization performance.

3. [recommended] Add a throughput metric (completed objective evaluations per CPU-second) alongside utilization, so evaluation efficiency is visible separately from core busyness.

If the reviewer insists, we could do that (with a potential footnote / warning in the paper), but would argue against this (now, here). a) Asking how much overhead a certain optimizer uses / produces is a valid question. And directgly comparable (per task). Of course, optimizers with a high overhead can still perform better, when they eval at "smarter" points, the whole design paradigm of BO, but it is interesting to compare how much this incurs. Table 6 in the appendix now makes this clearly visible. b) "Throughput" could potentially be misleading as this depends now also on the runtime of the points being selected (which is usually being heterogenous).

–> We can do this, with such an explanation, but worry that this might send a wrong message.

4. [recommended] Reconcile the version mismatch (text: rush 1.2.0; Zenodo record title: rush 1.1.1) and document the RNG seeding used in the benchmark scripts.

Version 1.1.1 in the Zenodo title referred to the benchmark repository, not the `rush` package. We reran all benchmarks with `rush` 1.3.0. The record name "benchmark 2026 rush 1.3.0" identifies the package version used now (https://zenodo.org/records/22899537). Section 5.1 states now that fixed random seeds were used. It also describes the separately seeded initial designs shared across strategies and the fixed OpenML cross-validation splits. Fixed seeds do not ensure exact reproducibility in asynchronous execution. The order of results depends on timing (Mentioned in Section 5.1 now).

5. [recommended] Specify the surrogate model and acquisition function used for each of CL, ACBO, and ADBO in the LightGBM benchmark.

Section 5.1 now gives the configuration for all three strategies. They use the `mlr3mbo` defaults: a random forest surrogate, a confidence bound acquisition function, and local search for acquisition

optimization.

6. [optional] A direct comparison against DeepHyper or Optuna would settle how rush stands relative to the established Python decentralized frameworks.

A comparison with DeepHyper and Optuna would help assess `rush` across languages. This comparison is beyond the scope of the present software paper. It is a possible topic for a future benchmark study.

Reviewer 3

1. [required] Reconcile the version discrepancy between the text (rush 1.2.0) and the Zenodo artifact title ("rush 1.1.1"). State the exact version that produced Tables 1, 2, and 3, and archive that version alongside the paper.

Version 1.1.1 referred to the benchmark repository, not the `rush` package. We reran the benchmarks with `rush` 1.3.0. The record name "benchmark 2026 rush 1.3.0" identifies the package version used now (https://zenodo.org/records/22899537).

2. [required] Report the full software environment for both benchmark sections, including R version, Redis server version and build, and mirai, redux, and ranger versions. Provide a captured environment such as sessionInfo output or a lockfile.

Section 4.1 now reports R 4.6.0 and Redis 8.10.1 from conda-forge. The benchmark repository includes `setup_hpc.sh` to install these versions. The `renv.lock` file records the whole R package environment.

3. [required] Document seed and RNG handling for the BO benchmarks, and state explicitly that each task-algorithm combination was run once, or qualify the utilization figures as single-run measurements.

We reran the benchmark with five replications per task-algorithm combination. Tables 2 and 6 report means and standard deviations. Section 5.1 states now that fixed random seeds were used. It also describes the separately seeded initial designs shared across strategies and the fixed OpenML cross-validation splits. Fixed seeds do not ensure exact reproducibility in asynchronous execution. The order of results depends on timing (Mentioned in Section 5.1 now).

4. [recommended] Add a short description of how concurrent Redis operations are kept conflict-free (single-threaded execution, transactions, or Lua scripts), since the paper asserts conflict-free registration without giving the mechanism.

Section 2.7, on data storage, now describes the concurrency mechanism. Individual Redis commands execute on a single thread, making operations on the task sets and lists atomic. Operations involving several keys are grouped into `MULTI`/`EXEC` transactions that execute without interruption by other commands. For example, `$push_running_tasks()` adds task keys to the running-task set and the list of all tasks in a single transaction. Potentially competing state transitions, including those performed by `$finish_tasks()` and `$fail_tasks()`, use atomic Lua scripts to ensure that a task is not recorded in two states simultaneously.

5. [recommended] Add a caveat to the utilization results noting that, per Table 6, most of ADBO's Toptimization is redundant per-worker surrogate fitting, so high utilization does not by itself indicate more useful objective throughput.

We added a learner utilization column to table 6 and describe the learner utilization that separates evaluation utilization from the CPU utilization in section 5.2.

However, we believe that surrogate fitting is not "redundant" in the context of ADBO. All three strategies fit one surrogate per proposed configuration and this is part of the optimization process rather than overhead. We think this is likely a (partial) misunderstanding, but also a subtle point, so let us explain this here carefully. Let's start with the metric U. We measure the time where CPUs are active (surrogate fitting, acquisition optimization, and learner training, summed over workers) divided by the complete time of an optim trace. Is this is $< 100\%$ CPUs must have been idle. And compute resources have been "underused", which is bad. We think nobody is disagreeing here. What we take from the editors criticism (repeated by multiple reviewers) is: If this is close to 100% this is not necessarily "ideal" *if* computation is redundantly / unnecessarily replicated i.e. the "redundant" fitting of the surrogates on the archive. But the whole point of ADBO is that this surrogate fitting / acquisition function optimization is *not* redundant as it does not operate on the same state! Let's go through a few examples. Let's take 2 workers. Worker 1 reads the archive. It fits a surrogate and then proposes a point. This point is evaluated. We wait. While we wait worker 2 reads the archive. This now includes the proposed point of W1 – with an imputed fitness values. State is different. Now it could happen that W2 reads the archive while W1 still proposes. Even then the state or proposal differs: a) The acquisition function is randomized (random lambda parameter in LCB per worker) and there is randomness in the random forest and acquisition function optimizer. b) Even the training data of the surrogate model is most of the time different. We can – in a multi-worker setup – likely assume that SOME points are under eval currently. These are also then imputed randomly. We added a footnote to section 5.1 explaining why surrogate fitting in ADBO is not redundant.

We checked how often the workers fitted a surrogate on the same archive with no running points. In this case the workers would indeed see the same archive because nothing is imputed.
In the first iteration of each worker, this happens often because the workers see only the initial design. But each time a different configuration was proposed due to the randomness in the acquisition function. After the first iteration, we found one case in the 20 ADBO runs where four workers saw the same archive with no running points (First replication of ADBO on credit-g). So redundant work is unlikely to happen during the optimization process.

6. [recommended] Provide a small-scale, local-Redis reproducible example (a few workers) so readers without HPC access can reproduce the utilization pattern qualitatively.

We would in general do that, but we do not exactly see how this could generate the desired / a reasonable result: A local example with a few workers can demonstrate the functionality of `rush` / technical features of rush. These are already available in our package vignette, so people can quickly play around with the package. But a local benchmark would likely not reproduce the utilization advantage observed in our benchmarks. This advantage arises when the central controller cannot propose configurations fast enough to keep workers occupied. With few workers, the controller will keep pace with demand. A small example would therefore not provide a representative reproduction of the reported utilization pattern.

7. [optional] A direct comparison against DeepHyper or Optuna would help position rush across languages, though this can reasonably be deferred to future work.

A comparison with DeepHyper and Optuna would help assess `rush` across languages. This comparison is beyond the scope of the present software paper. It is a possible topic for a future benchmark study.

Extra

1. When repeating the BO benchmarks, we observed delayed worker startup for the `kddcup09_appetency` task. The main cause was the transfer of task data to each worker over the LRZ network. We reduced this overhead by storing the data on a shared file system through the existing `large_objects_path` option. Workers on the same node can then use cached file reads. This reduces repeated network transfers. The approach requires a shared file system accessible to all worker nodes. It can reduce startup time when network transfer is a bottleneck. With sufficient network bandwidth, the benefit may be small. We explain the `large_objects_path` option in section 2.7 and 6 now.
2. rush 1.3.0 introduced profiles. We explain them shortly after they are printed with `$worker_info`