



SOFTWARE · FOUNDING ARTICLE

rush: Scalable Asynchronous Distributed Computing via Shared State in R

Marc Becker*  and Bernd Bischl 

Department of Statistics, Ludwig-Maximilians-Universität München, Munich; Munich Center for Machine Learning (MCML), Munich

*Corresponding author: marc.becker@stat.uni-muenchen.de

ARTICLE INFO

Keywords

asynchronous computing
distributed computing
R package
Redis
Bayesian optimization
hyperparameter optimization

History

Received 4 Sep 2026
Final version 19 Oct 2026
Accepted 13 Oct 2026
Version 1

Handling editor

Matthias Templ

ABSTRACT

Many algorithms in statistics and machine learning can be parallelized in an asynchronous manner where workers need to communicate through shared state rather than execute independent tasks dispatched by a central controller. Especially in modern hyperparameter optimization and parallel black-box optimization with expensive objectives, this decentralized approach has become widespread, and several Python frameworks adopt it (e.g., Optuna, DeepHyper, and Hyperopt). However, all popular R packages for parallel computing follow a centralized controller-worker architecture that does not support this pattern. We present rush, an R package that provides a shared-state coordination layer for asynchronously parallelized iterative algorithms. rush uses a Redis database as a shared key-value store: workers read and write task data through the database and independently execute their own loops. The package provides a high-level API for managing tasks and their lifecycle, featuring sub-millisecond per-task overhead, robust error handling with automatic detection of lost workers, and efficient caching. rush optionally integrates with the mlr3 ecosystem, powering asynchronous optimization in the bbotk and mlr3tuning packages. We demonstrate the practical utility of rush by implementing asynchronous decentralized Bayesian optimization (ADBO) and benchmarking it on hyperparameter optimization of LightGBM across four datasets using 448 workers.

✓ Reproduced verified 2026-10-13

REPRODUCIBILITY RECORD

Result	Reported results reproduced. Code examples re-run; all 492 table values recomputed from the archived runs. The 448-worker runs and the timings were not repeated.	✓ measured
Entry point	cd code/slds-lmu-benchmark_2026_rush-78385fd && Rscript run_all.R	✓ executed
Environment	R 4.6.0, rush 1.3.0, Redis 7.0.15; rocker/r-ver:4.6.0 on colima, linux/aarch64	✓ measured
Archive	rush-scalable-asynchronous-distributed-computing-via-shared-state-in-r-v1-repro.zip · reviewed bundle: the author's archived release https://doi.org/10.5281/zenodo.22899537 , SHA-256 941482b13321e8d90d8330486b2ad782336adc8fb56e97cd9ac2dadad8cabb22	✓ computed
Code	https://github.com/slds-lmu/benchmark_2026_rush/	declared
Data	Open.	declared, route checked

CITE AS Becker, M., & Bischl, B. (2026). rush: Scalable Asynchronous Distributed Computing via Shared State in R. Journal of Reproducible Statistics, 1, 5. <https://doi.org/10.68365/jrepstat.2026.5>

1 Introduction

Many iterative algorithms in statistics and machine learning share a common computational structure: multiple workers operate concurrently, each maintaining local state and autonomously deciding what to compute next, while coordinating through a shared pool of (intermediate) results. This computational pattern has long-standing roots in the *blackboard architecture* (Nii, 1986) and the *tuple space* model (Gelernter, 1985), coordination paradigms in which autonomous processes communicate through a shared associative memory.

Frameworks for parallel computation have become extremely popular in modern machine learning in the context of expensive black-box optimization and hyperparameter optimization (HPO, Bischl et al., 2023). Here, algorithmic approaches span a wide spectrum: from synchronous batch methods (Ginsbourger et al., 2010), through asynchronous but centralized schedulers such as ASHA (Li et al., 2020)¹ and Vizier (Golovin et al., 2017), to fully decentralized approaches such as asynchronous decentralized Bayesian optimization (ADBO, Egelé et al., 2023), where each worker autonomously decides what to compute next.

In contrast to these decentralized approaches, existing R packages for parallel computing follow a fundamentally different design. Packages such as `parallel`, `future` (Bengtsson, 2021), `mirai` (Gao, 2025), `batchtools` (Lang et al., 2017), and `crew` (Landau, 2026) adopt a controller-worker architecture (Figure 1, left): a central process proposes tasks, dispatches them to workers, and collects results. These frameworks excel at parallelizing independent computations but do not support algorithms in which workers coordinate through shared state and autonomously decide what to compute next. Beyond expressiveness — some algorithms are simply more natural when each worker has the coordination logic locally — the decentralized design also has computational advantages. Planning and generating further downstream tasks can themselves incur non-negligible cost, and moving this work from a central node to the workers allows it to run in parallel and enables more efficient task spawning, especially when many workers are available.

To fill this gap, we present `rush`, an R package that provides a shared-state coordination layer for iterative algorithms. `rush` uses a Redis database as its shared store: workers communicate by reading and writing task data—inputs, outputs, and computational states—through the database, each independently executing its own loop (Figure 1, right). The package provides a high-level API for managing tasks and their lifecycle, featuring sub-millisecond per-task overhead, robust error handling with automatic detection of lost workers, and an efficient caching mechanism that minimizes database operations. `rush` integrates with the `mlr3` (Lang et al., 2019) ecosystem, powering asynchronous optimization in the `bbotk` (Becker, Richter, et al., 2026) and `mlr3tuning` (Becker et al., 2025) packages.

While `rush` is designed as a general coordination layer and can be used for a wide range of algorithms, we demonstrate its utility through a case study in black-box optimization — which was also our primary motivation for developing `rush`. We implement ADBO (Egelé et al., 2023) on `rush` and benchmark it across four datasets using 448 workers. Our experiments show that ADBO on `rush` achieves a CPU utilization above 93% in every run, compared to single-digit utilization for centralized approaches on short-running tasks.

1.1 Related work

The widespread adoption of multi-core processors in the 2000s created a growing need to utilize these resources effectively for computational tasks in R. The first packages to address this need were `snow` (Rossini et al., 2007) and `multicore`. With R version 2.14.0 (released in 2011), parallel computing capabilities were integrated into the base R system through the `parallel` package. The `parallel` package provides `mclapply()` and `parLapply()` as parallel versions of `lapply()` for multicore and cluster computing, respectively. Although widely used, both functions block the R session until all tasks have completed, preventing retrieval of partial results. Moreover, their static task allocation can lead to poor load balancing when tasks have heterogeneous runtimes.

¹ASHA is algorithmically decentralized, but its common implementation in Ray Tune uses a central scheduler.

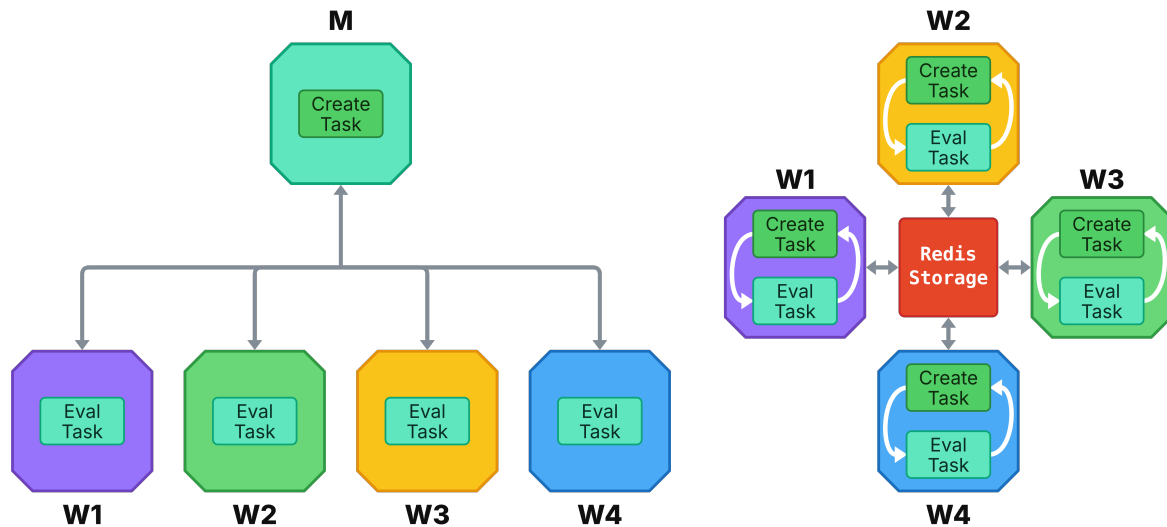


Figure 1 Centralized (left) and decentralized (right) networks. Octagons denote processes, including the main process *M* and workers *W*. In the centralized network, *M* proposes tasks to the workers and processes the returned computations. In the decentralized network, each worker *W* independently proposes tasks, evaluates them, and exchanges task information via a shared Redis database (red rectangle).

The *future* package (Bengtsson, 2021) introduced a unified parallel computing interface in R, supporting various backends such as *multisession*, *multicore*, and *callr*, with the *future.apply* package providing parallel versions of the **apply()* family of functions. However, the *multisession* and *cluster* backends are limited to 125 workers. Furthermore, per-task overhead — from process spawning (*multicore*, *callr*) or serialization across persistent workers (*multisession*, *cluster*) — can be significant for short-running tasks.

The *mirai* package (Gao, 2025) evaluates an R expression asynchronously in a parallel process, either locally or distributed across the network. The package has a strong focus on throughput, low overhead, and modern networking, eliminating many drawbacks of the older approaches. *mirai* provides a comprehensive daemon management interface for launching and configuring persistent daemons across local and remote systems (including SSH-based deployment and HPC schedulers). *rush* leverages this mechanism to start workers on remote machines.

The *batchtools* package (Lang et al., 2017) facilitates the execution of long-running tasks and large-scale computer experiments on HPC clusters. Because communication between the main process and workers relies entirely on the file system, the package incurs substantial per-task overhead from both job scheduling and file I/O, making it less suitable for high-throughput workloads with many short-running tasks. The more recent *crew* package (Landau, 2026) targets a similar use case, supporting long-running tasks in distributed systems ranging from traditional HPC clusters to cloud computing platforms, but likewise depends on a centralized controller for task coordination.

The parallel and distributed computing ecosystem in R is complemented by database connectivity layers that enable shared state and coordination. For relational database management systems, DBI (R Special Interest Group on Databases (R-SIG-DB) et al., 2024) defines a standardized interface that separates a common frontend from database-specific backends. For Redis, *redux* provides R bindings to the Redis API and exposes the command set through a low-level and a high-level interface.

The *rrq* (FitzJohn & Ashton, 2026) package is a task queue system for R using Redis as a coordination backend. It addresses the limitations of other packages by providing a non-blocking interface to parallel computing and keeping the overhead per task low. The package supports independent task queues, priority levels within queues, and inter-task dependencies. *rrq* is implemented on top of *redux*, which provides the underlying Redis connectivity and command interface. Its error-handling mechanism, particularly its heartbeat approach, directly informed the design of the corresponding mechanism in *rush*.

Python has a rich ecosystem for parallel and distributed computing. *Dask* (Rocklin, 2015) enables parallel and distributed execution via a dynamic task scheduler, and *Ray* (Moritz et al., 2018) is a general-

purpose distributed computing framework. For HPO specifically, many frameworks adopt a centralized architecture in which a central scheduler dispatches configurations to workers: Ray Tune (Liaw et al., 2018), Syne Tune (Salinas et al., 2022), Google Vizier (Golovin et al., 2017), Ax (Olson et al., 2025), Dragonfly (Kandasamy et al., 2020), and Keras Tuner (O'Malley et al., 2019). In contrast, frameworks such as Optuna (Akiba et al., 2019), DeepHyper (Egelé et al., 2025), Hyperopt (Bergstra et al., 2013), and Orion (Bouthillier et al., 2023) support decentralized operation by coordinating workers through a shared database (SQL, MongoDB, or Redis), enabling each worker to independently propose and evaluate configurations.

In summary, the existing R ecosystem for parallel and distributed computing is rich but uniformly follows a controller-worker architecture. Packages such as `parallel`, `future`, and `crew` centralize task proposal and result collection in a single process, which can become a bottleneck at scale. While `rrq` and `mirai` reduce per-task overhead, neither supports decentralized optimization loops in which workers independently propose and evaluate tasks. At least in the domains of expensive black-box optimization and HPO, Python offers frameworks such as DeepHyper and Optuna that support decentralized workflows, but no comparable solution exists in R. `rush` addresses this gap by combining a decentralized architecture with low-overhead Redis-based communication.

2 Architecture and API of `rush`

This section introduces the architecture and core API of `rush`. A `rush` network consists of multiple workers that communicate via a shared Redis database. A worker can read shared state from the database, compute new (partial) results, and potentially write them back into the shared database, as illustrated in Figure 2; see Figure 1 (right) for the full network topology.

2.1 Manager

The `Rush` manager class is responsible for creating, monitoring, and stopping a network. It is initialized using the `rsh()` function, which requires a network identifier and a `config` argument that specifies how to connect to the Redis database via the `redux` package. A running Redis server must be started separately beforehand.²

```
library(rush)

config = redux::redis_config()

rush = rsh(network = "demo-network", config = config)
```

Workers are started using the `$start_workers()` method, which accepts the worker loop (see Section 2.2) and the number of workers as arguments. Any additional named arguments are forwarded to the worker loop function. Workers run on `mirai` daemons (Gao, 2025), which are launched via `mirai::daemons()`.

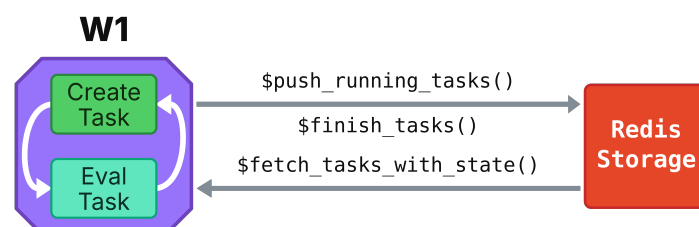


Figure 2 The communication flow between a worker and the Redis database in a `rush` network. The octagon represents a worker and the rectangle represents the Redis database. In each iteration, the worker reads shared state via `$fetch_tasks_with_state()`, registers a new task via `$push_running_tasks()`, computes the result, and writes it back via `$finish_tasks()`.

²Installation instructions for all platforms are available at https://redis.io/docs/latest/operate/oss_and_stack/install/install-redis/.

Calling `mirai::daemons(n)` opens a listener socket on the host, spawns a lightweight dispatcher implemented in C, and launches `n` persistent R processes that dial into the dispatcher and enter a receive–evaluate–return loop. Note that while `rush` still uses this mechanism to set up workers, most of the communication is actually facilitated via the Redis database, e.g., worker loop arguments, task inputs, outputs, and states. Daemons can be spawned on the local machine, on remote nodes via SSH, or on HPC clusters via job schedulers such as SLURM or SGE — in all cases, the daemon process dials back to the host and the same `rush` worker loop runs transparently. Alternatively, workers can be started locally via the `processx` package using `$start_local_workers()`, or a standalone script can be generated with `$worker_script()` for manual deployment; the only requirement is that the worker can connect to the Redis database.

```
mirai::daemons(4)

rush$start_workers(
  worker_loop = worker_loop,
  n_workers = 4)
rush$wait_for_workers(4)
```

Since `$start_workers()` returns immediately, `$wait_for_workers(n)` blocks until `n` workers have registered in Redis. Printing the `Rush` object reports the number of running workers and a summary of task states across the network (see Section 2.3).

```
rush
```

```
-- <Rush> -----
* Running Workers: 4
* Queued Tasks: 0
* Running Tasks: 0
* Finished Tasks: 0
* Failed Tasks: 0
```

The `$worker_info` active binding returns a `data.table` (Barrett et al., 2026) with one row per worker, containing the worker identifier, the process ID, the host on which it runs, whether a heartbeat (see Section 2.5) is active, and its current state.

```
rush$worker_info
```

	worker_id	pid	hostname	profile	heartbeat	state
	<char>	<int>	<char>	<char>	<lgl>	<char>
1:	pseudopolitic...	37025	ThinkPad-T14-...	<NA>	FALSE	running
2:	idiocratic_ca...	37028	ThinkPad-T14-...	<NA>	FALSE	running
3:	papery_sloth_...	37023	ThinkPad-T14-...	<NA>	FALSE	running
4:	precarnival_b...	37021	ThinkPad-T14-...	<NA>	FALSE	running

The `profile` column identifies each worker's compute profile, which can be used to group workers by available resources, e.g., CPUs or GPUs.

The workers can be stopped individually via `$stop_workers(ids)` or all together using the `$reset()` method, which also resets the network and database.

2.2 Worker loop

A worker loop is a plain R function that receives a `RushWorker` object for communicating with Redis.

```
worker_loop = function(rush, n_evals) {
  while (rush$n_finished_tasks < n_evals) {
    archive = rush$fetch_tasks_with_state(states = c("running", "finished"))
    xs = compute_task_inputs(archive)
```

```

    keys = rush$push_running_tasks(xss = list(xs))
    ys = compute_task_results(xs)
    rush$finish_tasks(keys, yss = list(ys))
  }
}

```

The above shows an abstract template for an exemplary situation where `rush` excels — and which is already fairly close to the asynchronous, decentralized HPO use case that we will cover in more detail in Section 3. In the (common) example above, we assume the following:

- The worker needs access to all available partial results to define and compute its next task.
- The inputs for the task also have to be computed and thus incur non-negligible costs.
- The computation of the task itself is time-consuming.

Inside the loop, the worker communicates with the shared Redis database through the following core methods of the `RushWorker` object:

- `$push_running_tasks(xss)` - Creates one or more new tasks with inputs `xss` in the database, marks them as "running", and returns their unique keys. Called before evaluation begins.
- `$fetch_finished_tasks()` - Retrieves all finished tasks from the database as a `data.table` (keys, inputs, outputs). Supports caching: previously fetched tasks are served from a local cache, and only new results are read from Redis.
- `$fetch_tasks_with_state(states)` - More general variant that retrieves tasks in specified states (e.g., both "running" and "finished"), useful when the worker needs to account for evaluations still in progress.
- `$finish_tasks(keys, yss)` - For tasks named by keys, writes corresponding results `yss` to the database and marks them as "finished".
- `$n_finished_tasks` - Active binding that returns the current global count of finished tasks across all workers, commonly used as a termination criterion.

Both `$push_running_tasks()` and `$finish_tasks()` accept an extra argument for associating auxiliary information with tasks and results that does not naturally fit into the inputs or outputs.

2.3 Tasks

Tasks are the basic units through which workers exchange information. Each task consists of four components: a unique key used to reference the task in the Redis database, a computational state, an input (`xs`), and an output (`ys`). The input and output are lists that may contain arbitrary data.

Tasks always have an associated computational state: "running", "finished", "failed", or "queued". The typical task lifecycle involves two transitions. The `$push_running_tasks()` method creates tasks marked as "running" and returns their keys. Upon completion, `$finish_tasks()` marks tasks as "finished" and stores the associated results. Tasks that encounter errors can be marked as "failed" using the `$fail_tasks()` method (see Section 2.5). The fourth state, "queued", supports a queue mechanism described in Section 2.4.

2.4 Queues

In addition to the autonomous loop pattern, `rush` supports a queue mechanism for cases in which tasks are created centrally and distributed to workers. The `$push_tasks()` method creates tasks in the "queued" state:

```

xss = lapply(1:20, function(i) {
  list(x1 = runif(1, -5, 5), x2 = runif(1, -5, 5))
})

rush$push_tasks(xss = xss)

```

Workers consume queued tasks using `$pop_task()`, which retrieves the next task, marks it as "running", and returns it. When the queue is empty, the method returns `NULL`, which serves as the termination signal:

```
worker_loop_queue = function(rush, fun) {
  repeat {
    task = rush$pop_task()
    if (is.null(task)) break
    ys = list(y = fun(task$xs$x1, task$xs$x2))
    rush$finish_tasks(task$key, yss = list(ys))
  }
}
```

The two patterns can be combined: workers first drain a queue of centrally created tasks, then transition to an autonomous loop for the remainder of the computation.

2.5 Error handling

rush provides error-handling mechanisms for two failure modes: R errors during task evaluation and unexpected worker crashes. For errors that occur during task evaluation, users must catch them within the worker loop and mark the corresponding task as "failed" using `$fail_tasks()`. If the error is not caught, the worker will crash.

```
tryCatch({
  ys = list(y = fun(xs$x1, xs$x2))
  rush$finish_tasks(key, yss = list(ys))
}, error = function(e) {
  condition = list(message = e$message)
  rush$fail_tasks(key, conditions = list(condition))
})
```

When a worker fails unexpectedly due to a crash, segmentation fault, or lost network connection, its tasks may remain in the "running" state indefinitely. The `$detect_lost_workers()` method identifies such workers and updates their state to "terminated". For workers started with `$start_workers()` or `$start_local_workers()`, lost worker detection works automatically by checking process status. Workers started via `$worker_script()` require an additional heartbeat mechanism: a background process periodically refreshes a Redis key with a set expiration timeout, and if the worker fails, the key expires, signaling that the worker is lost:

```
script = rush$worker_script(
  worker_loop = worker_loop,
  heartbeat_period = 1,
  heartbeat_expire = 3)
cat(strwrap(script, width = 76, exdent = 2), sep = "\n")
```

```
Rscript -e 'rush::start_worker(network_id = "demo-network", config =
  list(scheme = "redis", host = "127.0.0.1", port = "6379"),
  heartbeat_period = 1L, heartbeat_expire = 3L)'
```

2.6 Logging

Workers can write messages generated via the `lgr` (Fleck, 2026) package to the database. The `lgr_thresholds` argument of `$start_workers()` specifies the logging level for each logger by name, e.g., `c("mlr3/rush" = "debug")`. The `mlr3/` prefix denotes the parent logger shared across the `mlr3` ecosystem, into which `rush` integrates. Logging introduces a minor performance overhead and is disabled by default.

```
wl_log_message = function(rush) {
  lg = lgr::get_logger("mlr3/rush")
  lg$info("This is an info message from worker %s", rush$worker_id)
}
```

```

rush$start_workers(
  worker_loop = wl_log_message,
  n_workers = 2,
  lgr_thresholds = c("mlr3/rush" = "info"))

```

To retrieve all log entries, use the `$read_log()` method.

```

Sys.sleep(1)

rush$read_log()[, list(worker_id, msg)]

```

	worker_id <char>	msg <char>
1:	ungrammatical_beauceron_32fe...	[rush] This is an info messa...
2:	accessible_bear_15e5052b	[rush] This is an info messa...

Outputs and messages can be written to files. Set a path to a folder via the `message_log` and `output_log` arguments of `$start_workers()`. Messages and outputs are written to files named `message_<worker_id>.log` and `output_<worker_id>.log` in that folder, respectively.

```

rush$start_workers(
  worker_loop = wl_log_message,
  n_workers = 2,
  message_log = "logs",
  output_log = "logs")

```

2.7 Data storage

By default, `rush` stores all data in a Redis database. Redis is an open-source, in-memory key-value store commonly used as a database, cache, and message broker. It offers extremely low latency and supports multiple data structures such as strings, hashes, lists, and sets. `rush` maps its data model onto three Redis data structures: hashes, sets, and lists.

Redis hashes are a data structure for storing associative field–value pairs under a single key. They are optimized for memory efficiency and fast access to structured data. This makes them suitable for representing the input, output, and metadata of tasks. The key of the hash identifies the task in Redis and `rush`. A task is stored as a Redis hash with the following structure:

```
key : xs | ys | xs_extra
```

The field–value pairs are written by different methods, e.g., `$push_running_tasks()` writes `xs` and `$finish_tasks()` writes `ys`. The values of the fields are serialized lists or atomic values, e.g., deserializing the `xs` field gives `list(x1 = 1, x2 = 2)`. This structure allows a hash to be efficiently converted into a table row and multiple hashes to be joined into a table. When retrieved, multiple hashes are joined into a tabular format:

	key	x1	x2	y	timestamp
1..	3	4	7	12:04:11	
2..	1	4	5	12:04:12	
3..	1	1	2	12:04:13	

Redis sets are unordered collections of unique elements that support efficient membership tests. `rush` uses sets to track task states; for example, the keys of all currently running tasks are stored in the `"running_tasks"` set.

Lists in Redis are ordered collections stored as linked lists that support efficient insertion and removal at both ends. `rush` uses lists for two purposes: managing the task queue and recording finished tasks. For the queue, the list structure is a natural choice, as it supports atomic push and pop operations. Storing

the finished tasks in a list gives them an order by time, enabling workers to cheaply retrieve only the most recent results.

`rush` uses three mechanisms to coordinate concurrent access to the database. First, Redis executes commands on a single thread, making individual operations such as adding a key to a set or appending a key to a list atomic. Second, operations involving several keys are grouped into MULTI/EXEC transactions, which Redis executes without interruption by other commands. For example, `$push_running_tasks()` adds new keys to the "running_tasks" set and the list of all tasks in a single transaction. Third, potentially competing state transitions are implemented as atomic Lua scripts. These include transitions by `$finish_tasks()` and `$fail_tasks()`, insuring that a task is not recorded in two states simultaneously.

To minimize the number of Redis round-trips when workers repeatedly access many tasks (common in iterative algorithms), each `Rush` instance keeps a local `data.table` cache of the finished tasks it has already seen. On subsequent fetches, only the keys after the current cache length are requested; the corresponding hashes are read and appended to the cached table. This way, repeated queries scale with the number of newly finished tasks instead of the total result history.

The `large_objects_path` option allows large objects to be stored on a shared file system accessible to all workers. This can reduce network transfer overhead by allowing workers on the same node to benefit from cached file reads.

2.8 Retrieving results

Once all computational tasks are completed, the results can be retrieved from the database. As already explained, the `$fetch_finished_tasks()` method returns a `data.table` containing the task key, input, and result.

2.9 Testing

`rush` includes 105 unit tests written with `testthat` (Wickham, 2011). GitHub Actions runs `R CMD check`, including these tests, on Ubuntu with 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.

3 Ask-and-tell and asynchronous decentralized Bayesian optimization

3.1 The ask-and-tell pattern in optimization

The worker loop we discussed in Section 2.2 directly mirrors the *ask-and-tell* interface pattern that has become the dominant API design for black-box optimization frameworks. In this pattern, the optimizer exposes two methods — `$ask()`, which proposes the next candidate point(s), and `$tell()`, which receives the corresponding evaluation result(s) — while the user retains full control over the evaluation loop. In the worker loop template, the trio of `$fetch_tasks_with_state()`, `compute_task_inputs()`, and `$push_running_tasks()` corresponds to the *ask* step, while `$finish_tasks()` corresponds to the *tell* step. This design decouples the optimizer from the objective function evaluation, allowing users to integrate optimization into arbitrary workflows. Prominent implementations include `Optuna` (`study.ask()/study.tell()`, Akiba et al., 2019), `Ax` (`get_next_trial()/complete_trial()`, Olson et al., 2025), `SMAC3` (Lindauer et al., 2022), and `scikit-optimize` (Head et al., 2018).

The ask-and-tell interface enables two fundamentally different parallelization strategies for optimization, which differ in where the optimization logic runs and how workers coordinate.

Synchronous batch parallelism. In this scenario, a single central process runs the optimizer. In each iteration, the optimizer asks for a batch of q candidate points, which are then evaluated in parallel across q workers using a standard parallel backend (e.g., `future`, `mirai`, or `parallel`). Once all q evaluations complete, the results are told back to the optimizer, which updates its model and proposes the next batch. No shared state beyond the optimizer is needed, but depending on the complexity of the `$ask()` step, the central optimizer can become a bottleneck. Furthermore, synchronous batches waste resources when

evaluation times are heterogeneous, as all workers must wait for the slowest evaluation in each batch to complete.

Asynchronous decentralized parallelism. In this scenario, each worker runs its own local optimization and ask-and-tell loop: it reads the shared history of completed (and possibly running) evaluations, asks its local optimizer for a proposal, evaluates it, and tells the result back to a shared store. Workers operate fully independently — they do not wait for each other and do not coordinate through a central controller. This eliminates both the controller bottleneck and the synchronization barrier. However, it requires infrastructure for shared-state coordination: workers must be able to read each other's results and register their own tasks in a concurrent, conflict-free manner.

3.2 Asynchronous decentralized Bayesian optimization

Bayesian optimization (BO) is a sample-efficient strategy for optimizing expensive black-box functions. It maintains a probabilistic surrogate model (typically a Gaussian process or random forest) fitted to all completed evaluations, and selects the next evaluation point by optimizing an acquisition function that balances exploration and exploitation. In the standard sequential setting, a single process alternates between fitting the surrogate, proposing a point, and evaluating it. In ADBO (Egelé et al., 2023), this loop is replicated across multiple workers, each running independently. Every worker reads the shared history of evaluations from the database, fits its own local surrogate model, and proposes the next point autonomously — without waiting for other workers or coordinating through a central controller. For design points currently being evaluated on other workers (i.e., without a known objective value yet), a fake objective is imputed — usually a value between the minimum and maximum observed objective values in the archive. This penalizes the proposal of further design points near those currently under evaluation. Results are written back to the shared store as they complete, so all workers benefit from each other's evaluations. To encourage diversity among proposals, each worker uses a stochastic acquisition function with different exploration–exploitation trade-offs (e.g., different λ values in a confidence bound criterion). This architecture eliminates the central bottleneck that arises in synchronous or centralized parallel BO, where a single process must fit the surrogate and propose points for all workers; see Section 5.

We use the Branin function, defined over the domain $x_1 \in [-5, 10]$ and $x_2 \in [0, 15]$, as a toy objective:

```
branin = function(x1, x2) {
  (x2 - 5.1 / (4 * pi^2) * x1^2 + 5 / pi * x1 - 6)^2 +
  10 * (1 - 1 / (8 * pi)) * cos(x1) + 10
}
```

Next, we define an optimizer that proposes new points for evaluation. It takes as input an archive of running and completed tasks and returns a new candidate point x_s for evaluation. The archive is represented as a `data.table` object. If the archive is empty, the optimizer proposes a random point. Otherwise, it imputes missing objective values for running tasks using the mean objective value of completed tasks. It then fits a random forest surrogate model with the `ranger` (Wright & Ziegler, 2017) R package and proposes a new point by minimizing a lower confidence bound acquisition function $\hat{\mu}(x) - \lambda \hat{\sigma}(x)$. The exploration parameter λ is passed as an argument; each worker draws its own value from $\text{Exp}(1)$ once at the start of the worker loop, following Egelé et al. (2023). Assigning each worker a different λ realizes the diversification mechanism of ADBO: independent workers explore different exploration–exploitation trade-offs without any coordination. Fitting the surrogate model becomes increasingly expensive as the number of tasks in the archive grows. A detailed understanding of this optimizer is not necessary to follow the subsequent examples. Two properties are relevant for what follows: it uses previously evaluated points from other workers to inform new proposals, and proposal generation incurs a non-negligible computational cost.

```
optimizer = function(archive, lambda) {
  if (nrow(archive) == 0) {
    return(list(x1 = runif(1, -5, 10), x2 = runif(1, 0, 15)))
  }
}
```

```

mean_y = mean(archive$y, na.rm = TRUE)
archive["running", y := mean_y, on = "state"]

surrogate = ranger::ranger(
  y ~ x1 + x2,
  data = archive,
  num.trees = 100L,
  keep.inbag = TRUE)

xdt = data.table::data.table(
  x1 = runif(1000, -5, 10),
  x2 = runif(1000, 0, 15))
p = predict(surrogate, xdt, type = "se", se.method = "jack")
cb = p$predictions - lambda * p$se

as.list(xdt[which.min(cb)])
}

```

The BO worker loop combines the optimizer with the rush API. Note that it uses `$fetch_tasks_with_state()` to retrieve both running and finished tasks, allowing the optimizer to impute values for evaluations still in progress. This is the ADBO pattern: each worker fits its own surrogate model to all available data and independently proposes the next configuration.

```

workerloop_adbo = function(rush, branin, optimizer) {
  # evaluate the initial design
  repeat {
    task = rush$pop_task()
    if (is.null(task)) break
    ys = list(y = branin(task$xs$x1, task$xs$x2))
    rush$finish_tasks(task$key, yss = list(ys))
  }

  # draw the worker-specific exploration parameter
  lambda = rexp(1)

  # ADBO loop
  while (rush$n_finished_tasks < 100) {
    archive = rush$fetch_tasks_with_state(states = c("running", "finished"))
    xs = optimizer(archive, lambda)

    key = rush$push_running_tasks(xss = list(xs))

    ys = list(y = branin(xs$x1, xs$x2))
    rush$finish_tasks(key, yss = list(ys))
  }
}

```

In practice, the optimization often starts with an initial design — a set of space-filling points that seed the surrogate model. The queue mechanism from Section 2.4 provides a natural way to distribute such an initial design: a structured design (e.g., Latin hypercube sampling) is created once with the `lhs` (Carnell, 2026) R package and pushed to the queue, and workers drain it before transitioning to the model-based loop.

```

lhs_points = lhs::maximinLHS(n = 25, k = 2)
x1_lower = -5
x1_range = 15
x2_lower = 0
x2_range = 15

xss = lapply(1:25, function(i) {

```

```
# rescale to the domain
list(
  x1 = lhs_points[i, 1] * x1_range + x1_lower,
  x2 = lhs_points[i, 2] * x2_range + x2_lower)
})

rush$push_tasks(xss = xss)
```

4 Benchmarking of core functions

4.1 Experimental setup

The evaluation of short-running tasks is sensitive to the overhead introduced by `rush`. To quantify this overhead, we benchmark the core functions: `$push_running_tasks()`, `$finish_tasks()`, and `$fetch_finished_tasks()`. For `$push_running_tasks()` and `$finish_tasks()`, we vary the number of parameters (1, 10, 100) of the task and the payload size (1, 10, 100, 1,000, 10,000) of each parameter. In the case of `$finish_tasks()`, the payload size refers to the result. The payload size is the number of elements in a double vector where each double takes 8 bytes, i.e., a payload size of 10,000 corresponds to 80 kB. We benchmark `$fetch_finished_tasks()` both with and without caching to measure the effect of the caching mechanism. In addition to the number of parameters and payload size, we vary the number of tasks (1, 10, 100, 1,000, 10,000, 100,000). When caching is enabled, the cache holds all previously fetched tasks except the most recent one, so that each call retrieves only a single new result from the database. For both `$fetch_finished_tasks()` benchmarks, we cap the payload size at 100 to avoid creating overly large tables in memory. All benchmarks use `rush` version 1.3.0 with R 4.6.0 and Redis 8.10.1. All measurements are taken by a single worker against a local Redis instance communicating via a Unix socket.³ The runtime is measured with the `microbenchmark` (Mersmann et al., 2024) R package. We report the median and the 95th percentile over the replicates. All figures are created with the `ggplot2` (Wickham, 2016), `cowplot` (Wilke, 2025), and `scales` (Wickham et al., 2025) R packages.

³The multi-node BO benchmarks in Section 5 use TCP, which adds some overhead.

4.2 Results

Table 1 Runtime of `$push_running_tasks()` and `$finish_tasks()` as a function of the number of fields and payload size. For `$push_running_tasks()`, the number of fields corresponds to the number of parameters and for `$finish_tasks()`, it corresponds to the number of results. The runtime is reported as the median over 10,000 replicates with the 95th percentile in parentheses.

Number of Fields	Payload Size	Runtime Push (ms)	Runtime Finish (ms)
1	1	0.34 (0.75)	0.26 (0.39)
1	10	0.33 (0.62)	0.25 (0.34)
1	100	0.33 (0.73)	0.26 (0.42)
1	1,000	0.34 (0.61)	0.26 (0.32)
1	10,000	0.55 (1.27)	0.34 (0.49)
10	1	0.33 (0.39)	0.25 (0.32)
10	10	0.41 (1.15)	0.25 (0.33)
10	100	0.44 (1.15)	0.29 (0.38)
10	1,000	0.39 (1.16)	0.35 (0.51)
10	10,000	1.89 (2.58)	1.33 (2.03)
100	1	0.45 (1.11)	0.27 (0.46)
100	10	0.53 (1.32)	0.28 (0.36)
100	100	0.58 (1.30)	0.36 (0.48)
100	1,000	0.92 (1.45)	1.37 (2.07)
100	10,000	12.98 (25.41)	12.07 (20.80)

The runtime of the functions `$push_running_tasks()` and `$finish_tasks()` is summarized in Table 1. For payload sizes up to 100, pushing a task to the database takes a median of 0.33–0.58 ms, and finishing a task takes slightly less, 0.25–0.36 ms. For the largest configuration (payload size 10,000 and 100 fields), the median push and finish times increase to 12.98 ms and 12.07 ms, respectively. The corresponding 95th percentiles are 25.41 ms and 20.80 ms, approximately 2.0 and 1.7 times the medians, indicating greater upper-tail latency for large payloads. For realistic HPO workloads, which typically involve tens of hyperparameters with scalar values (payload size 1), the median per-task overhead remains well below 1 ms.

For fetching, `rush` provides a built-in caching mechanism. The difference between fetching with and without caching is negligible for a small number of tasks. However, as the archive grows beyond 1,000 tasks, caching yields a substantial speedup, as shown in Figure 3 and Table 3. At 10,000 tasks and 10 parameters, fetching without the cache takes approximately 556 ms, compared to 3 ms with caching.

5 Benchmarks of Bayesian optimization

5.1 Experimental setup

We compare the effective CPU utilization of three parallelization strategies for BO on HPO tasks, representing increasing degrees of decentralization. We optimize nine hyperparameters of `LightGBM` (Ke et al., 2017), using the implementation in `mlr3extralearners` (Fischer et al., 2025), on four datasets, as summarized in Table 4 and Table 5. The datasets are downloaded from OpenML with `mlr3oml` (Lang & Fischer, 2026) and differ in their number of features and observations to induce a range of training runtimes. Each experiment runs for 10 minutes on 448 workers. The benchmarks are conducted on a Linux cluster at the Leibniz Supercomputing Centre (LRZ). Each node has 112 Intel Xeon Sapphire Rapids cores and 488 GiB of memory, running SUSE Linux Enterprise Server 15 SP6. We use four nodes for the benchmarks, resulting in 448 cores in total. All benchmarks use `rush` version 1.3.0.

Synchronous batch BO (CL). Evaluates q hyperparameter configurations in parallel using the constant liar (CL) approach (Ginsbourger et al., 2010). The first point is obtained by maximizing the acquisition function. A fake value is then imputed, the surrogate model is updated, and the next point is

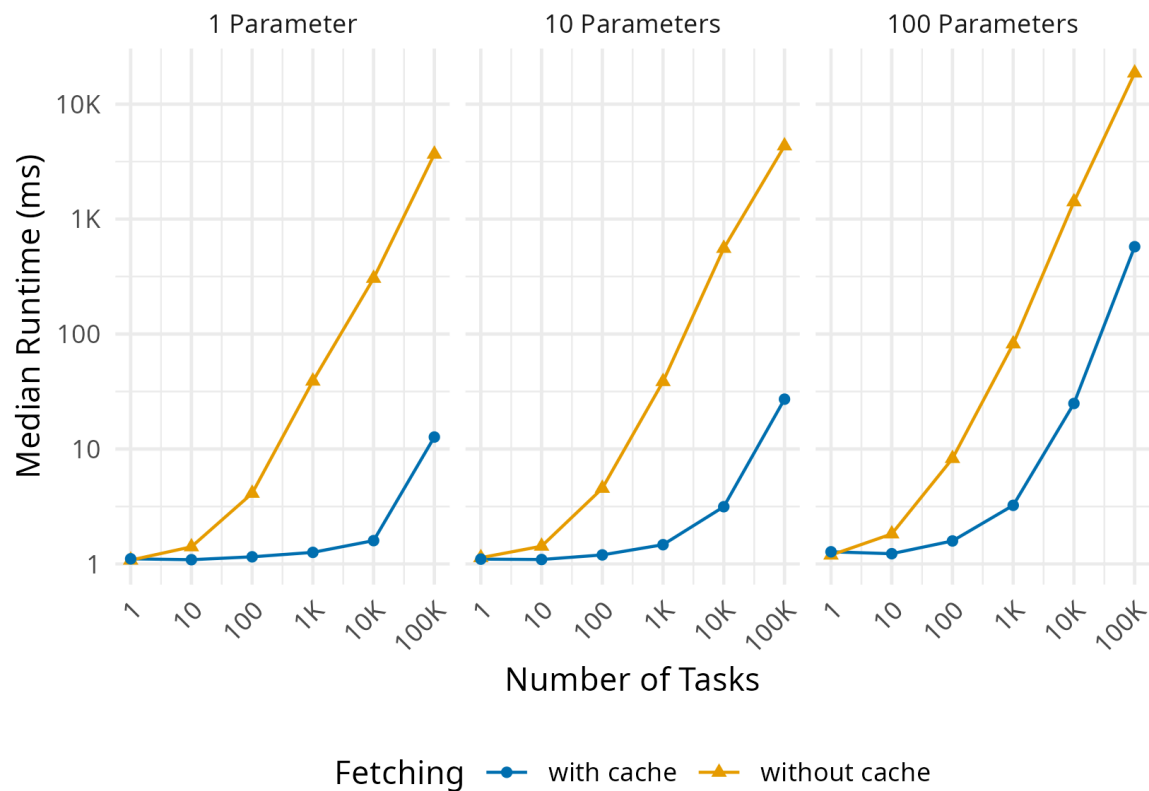


Figure 3 Median runtime in milliseconds of fetching finished tasks with (blue circles) and without (orange triangles) caching for different numbers of parameters as a function of the total number of tasks in the database. Both axes are log-scaled. The payload size is 1 in this benchmark. Without caching, all tasks are retrieved from the database on each call. With caching, only one new task is retrieved from the database; all previously fetched tasks are served from the worker's cache.

proposed by maximizing the acquisition function again. Common imputation choices are the minimum, maximum, or predicted posterior mean of already evaluated points. In our setup, CL proposes 44 configurations evaluated with 10-fold cross-validation, resulting in 440 parallel evaluations. The remaining 8 cores cannot be used by design, since each configuration occupies 10 cores. `mlr3` uses `mirai` to parallelize over the resampling iterations. Batch parallelization requires all workers to finish before the next batch can be proposed. We use early stopping to determine the optimal number of boosting iterations, which naturally induces heterogeneous runtimes: some models train for only a few iterations while others run for up to 5,000. This heterogeneity exposes the synchronization overhead inherent in batch approaches, as fast workers remain idle while waiting for slower ones to complete.

Asynchronous centralized BO (ACBO). Eliminates synchronization overhead by evaluating configurations asynchronously. When a worker becomes idle, the central controller proposes a new configuration and sends it to the worker without waiting for other evaluations to complete. Objective values for configurations currently under evaluation are imputed, as in CL. However, surrogate fitting and acquisition function optimization remain centralized and are executed sequentially by the controller. With many workers, this sequential bottleneck limits scalability: new configurations cannot be proposed fast enough to keep all workers occupied.

Asynchronous decentralized BO (ADBO). Distributes both surrogate updates and acquisition function optimization across the workers. Each worker maintains its own surrogate model and independently proposes the next configuration. When a worker completes an evaluation, it asynchronously shares the result via the database; other workers incorporate this information into their local models. As in CL and ACBO, objective values for configurations currently being evaluated are imputed. ADBO additionally uses stochastic acquisition functions to promote varying exploration–exploitation trade-offs between

workers, e.g., by initializing the λ parameter of the confidence bound acquisition function with different values for each worker.⁴ ADBO requires infrastructure that allows workers to exchange information asynchronously without a central controller — the architecture provided by *rush*.

All three strategies are implemented with *mLr3tuning* (Becker et al., 2025) and *mLr3mbo* (Becker, Schneider, et al., 2026) and use the *mLr3mbo* defaults: a random forest surrogate model (Wright & Ziegler, 2017), a confidence bound acquisition function, and local search for acquisition function optimization. The strategies differ in the number of cores used per configuration. CL distributes the 10 resampling iterations across 10 cores and evaluates 44 configurations in parallel. ACBO and ADBO execute the resampling iterations sequentially on a single core, allowing up to 448 configurations to be evaluated concurrently. For each task and replication, we randomly sample an initial design of 100 configurations from the search space in Table 4 and evaluate them using 10-fold cross-validation. The evaluation counts, wall-clock times, and utilization in Table 2 and the traces in Figure 5 exclude the initial design. Each task–algorithm combination is run five times. Benchmark jobs use fixed random seeds, with initial designs shared across strategies and fixed OpenML cross-validation splits. In asynchronous execution, result ordering depends on timing, so fixed seeds alone do not ensure exact reproducibility.

We measure effective CPU utilization following Egelé et al. (2023). Let n_{workers} denote the number of workers, where each worker corresponds to a single CPU core, and T_{walltime} be the elapsed wall-clock time. The total available CPU time is $T_{\text{CPU}} = T_{\text{walltime}} \cdot n_{\text{workers}}$. Let $T_{\text{optimization}}$ denote the cumulative time spent on surrogate fitting, acquisition optimization, and model training, summed across all workers. The effective CPU utilization is then $U = \frac{T_{\text{optimization}}}{T_{\text{CPU}}}$. Utilization thus counts the time spent proposing configurations as well as evaluating them. All three strategies fit one surrogate and optimize the acquisition function once per proposed configuration. To separate the two components, Table 6 additionally reports the learner utilization, which counts only the time spent evaluating configurations.

Table 2 Effective CPU utilization and the number of completed evaluations for CL, ACBO, and ADBO across four benchmark tasks. For each task–algorithm combination, the table reports the mean runtime of the 10-fold cross-validation (CV) in seconds, the number of completed evaluations, and the resulting effective CPU utilization in percent. All entries are the mean over five replications with the standard deviation in parentheses. Because each algorithm follows a different search trajectory, the configurations evaluated differ, leading to variation in mean CV runtimes across algorithms on the same dataset. This complicates direct comparison of raw evaluation counts: e.g., on airlines, ACBO’s mean evaluation (158 s) is shorter than CL’s (286 s), so part of ACBO’s higher evaluation count reflects faster individual evaluations rather than only better scheduling. The same effect explains the large standard deviation of the evaluation count of ADBO on airlines, whose mean evaluation ranges from 171 s to 403 s across replications.

Task	Algorithm	Mean Runtime CV (s)	Evaluations	Utilization (%)
credit-g	CL	0.65 (0.06)	502 (24)	0.3 (0.0)
	ACBO	0.53 (0.02)	545 (13)	0.3 (0.0)
	ADBO	0.71 (0.02)	15,796 (231)	94.6 (0.1)
KDDCup09_appetency	CL	26.46 (2.71)	132 (0)	1.1 (0.1)
	ACBO	23.21 (0.94)	526 (7)	4.9 (0.2)
	ADBO	30.05 (0.80)	6,520 (106)	93.7 (0.1)
adult	CL	11.53 (0.74)	370 (24)	1.8 (0.1)
	ACBO	11.43 (2.69)	500 (15)	2.4 (0.6)
	ADBO	15.12 (0.33)	9,728 (108)	95.7 (0.0)
airlines	CL	285.90 (73.31)	97 (20)	9.2 (1.6)
	ACBO	157.83 (29.29)	334 (31)	31.9 (7.6)
	ADBO	270.85 (95.37)	494 (365)	99.5 (0.2)

⁴The workers therefore do not repeat each other’s computations: their archives usually differ in the configurations under evaluation and the imputed values, and even on identical archives, the randomness in the surrogate, the acquisition function, and its optimizer leads to different proposals.

5.2 Results

The effective CPU utilization and the number of completed evaluations are summarized in Table 2 as means and standard deviations over the five replications. Across all tasks and all replications, ADBO achieves substantially higher utilization than both CL and ACBO, and this improvement is consistent across datasets with both short and long training runtimes. The utilization of ADBO varies by less than one percentage point between replications of the same task, so the standard deviations in Table 2 are small compared to the differences between the algorithms. ADBO also completes substantially more evaluations within the same wall-clock budget. Utilization includes the surrogate fitting needed to propose configurations: on credit-g, ADBO spends a mean cumulative 211,614 s fitting surrogates versus 11,164 s evaluating configurations (Table 6). ADBO spends more time per proposal on surrogate fitting than CL and ACBO because the cost of fitting a random forest grows with the size of the archive, which reaches over 15,000 evaluations for ADBO but stays below 650 for CL and ACBO. Counting only the time spent evaluating configurations, ADBO reaches a mean learner utilization of 4.2% on credit-g, compared to 0.1% for CL and ACBO, and between 56.2% and 99.2% on the other tasks, compared to at most 31.7% for CL and ACBO (Table 6).

All three methods reach similar final classification errors; ADBO achieves slightly lower mean errors, with differences of up to 0.8 percentage points on credit-g and less than 0.1 percentage points on the other tasks. High utilization alone therefore does not imply better optimization performance. The additional evaluations mainly help reach a given error earlier (Figure 5).

On tasks with very short training runtimes, centralized methods leave most compute resources idle. This effect is most pronounced on the credit-g task, where CL and ACBO complete only a few hundred evaluations while ADBO completes over 15,000 in every replication. The utilization of ADBO is between 270 and 300 times that of CL and ACBO across the five replications. Figure 4 illustrates the underlying mechanism. During the proposal phase, only a single core is busy, and for short-running objectives this phase dominates the wall-clock time.

On medium-cost tasks such as adult and KDDCup09_appetency, the same pattern persists. ACBO improves utilization relative to CL by eliminating synchronization overhead, but the centralized surrogate update remains a limiting factor: on adult, for instance, ACBO achieves a mean utilization of 2.4% compared to 95.7% for ADBO.

The airlines task has the longest training times, which reduces the relative overhead of surrogate updates and consequently increases utilization for all algorithms. Utilization also varies most between replications on this task, since a run evaluates only a few hundred configurations and their cost differs strongly: the utilization of ACBO ranges from 20% to 41% and its mean evaluation runtime from 132 s to 202 s across the five replications. Even in this favorable regime, ADBO maintains a substantial advantage, achieving between 2.4 and 4.9 times the utilization of ACBO in the individual replications. The decentralized architecture therefore scales robustly across workload regimes.

Beyond utilization, the parallelization strategy also affects budget adherence. The wall-clock runtime of CL can overshoot the nominal 10-minute budget, sometimes substantially (e.g. on KDDCup09_appetency; see Table 6), because CL checks the termination criterion only after an entire batch finishes. If a batch starts shortly before the time limit, the experiment continues until all evaluations complete, introducing significant budget overruns. In contrast, ACBO and ADBO check for termination after every completed evaluation, enforcing the wall-clock constraint more accurately. Asynchronous scheduling therefore improves both utilization and budget adherence.

6 Discussion and conclusion

We presented *rush*, a package for asynchronous and decentralized optimization in R. Its database-centric architecture eliminates the central controller bottleneck by allowing workers to communicate through a shared Redis database and independently execute their own optimization loops. The package combines sub-millisecond per-task overhead with an efficient caching mechanism and integrates with the `mlr3` ecosystem as the backend for asynchronous optimization in `bbotk` and `mlr3tuning`. Our benchmarks

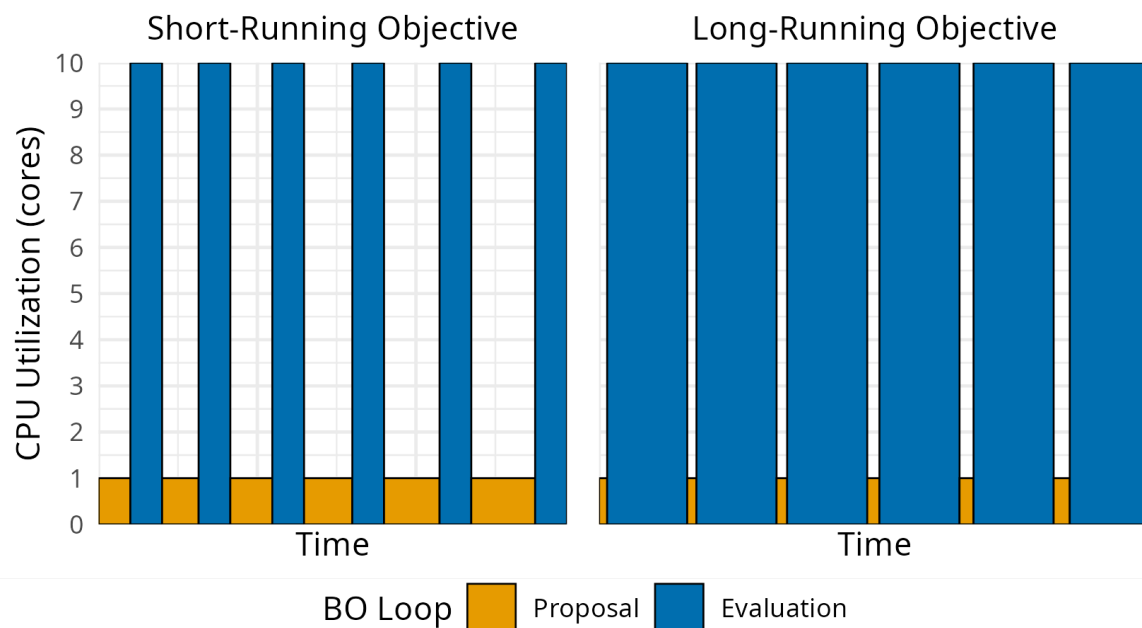


Figure 4 CPU utilization for short-running and long-running objective functions. Bars represent the proposal phase (light orange) and evaluation phase (dark blue) of the BO loop. Bar height indicates CPU utilization on a 10-core machine: during proposal, CL uses a single core, whereas during evaluation it uses all 10 cores. The proposal phase grows over iterations as surrogate fitting becomes more expensive with an increasing number of observations. For long-running objectives, evaluation dominates wall-clock time, yielding higher average CPU utilization than for short-running objectives. The effect is far more pronounced at the 448-worker scale used in our benchmarks, where the proposal phase occupies only a single worker, about 0.2% of the available resources, leaving the remaining 99.8% idle.

demonstrate that ADBO on rush achieves a CPU utilization above 93% on 448 workers in every run, compared to single-digit utilization for centralized approaches on short-running tasks. The sub-millisecond overhead measurements apply to a single worker using a local Unix socket; per-operation latency under concurrent TCP access was not measured separately.

Despite these results, several limitations should be noted. The benchmarks are designed to measure utilization and not final optimization performance. All three algorithms reach similar final classification errors within the 10-minute budget (Figure 5 and Table 6). ADBO reaches the lowest mean error on all four tasks, but the differences are small: 0.8 percentage points on credit-g and less than 0.1 percentage points on the other three tasks. The additional evaluations under ADBO mainly translate into reaching a given error earlier rather than into a lower final error. Whether the higher throughput yields better final solutions on more complex problems like hierarchical search spaces must be evaluated in future work. The caching mechanism reduces fetch overhead substantially, but the cost of binding new rows to the cached data.table grows with archive size, which may become relevant for optimization runs with very large numbers of evaluations. We do not provide a direct comparison with Python frameworks such as DeepHyper or Optuna, which would help position rush in the broader cross-language landscape. Worker startup can be delayed by transferring large datasets over the network, as observed for kddcup09_appetency on the LRZ cluster. Where a shared file system is available, the large_objects_path option can reduce this overhead through cached file reads (Section 2.7).

Several directions for future work emerge from these limitations. Adaptive caching strategies, such as pre-allocation, could mitigate the growing cost of row binding for very large archives. The queue mechanism could be extended to support more complex task-dependency graphs, enabling workflows in which tasks have prerequisites or conditional branching.

By providing a framework for decentralized optimization in R, rush enables a class of algorithms that was previously available only in the Python ecosystem. As computational budgets and model complexity continue to grow, the ability to scale optimization efficiently across large numbers of workers becomes

increasingly important. We hope that `rush` will serve as a foundation for the development and application of new distributed optimization algorithms in R.

Acknowledgments

The authors gratefully acknowledge the computational and data resources provided by the Leibniz Supercomputing Centre (www.lrz.de). We thank Martin Binder and Sebastian Fischer for their valuable feedback.

Competing interests

The authors develop and maintain the `rush`, `bbotk`, `mlr3tuning`, and `mlr3mbo` packages benchmarked in this paper. They declare no financial competing interests.

Data and Code Availability

The package is released on CRAN and its development version is hosted at <https://github.com/mlr-org/rush>. All code is available on zenodo at <https://zenodo.org/records/22899537> (Becker, 2026).

References

- Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2623–2631. <https://doi.org/10.1145/3292500.3330701>
- Barrett, T., Dowle, M., Srinivasan, A., Gorecki, J., Chirico, M., Hocking, T., Schwendinger, B., & Krylov, I. (2026). *data.table: Extension of data.frame* [Manual]. <https://doi.org/10.32614/CRAN.package.data.table>
- Marc Becker. (2026). *slds-lmu/benchmark_2026_rush: benchmark 2026 rush 1.3.0* [Computer software]. Zenodo. <https://doi.org/10.5281/ZENODO.22899537>
- Becker, M., Lang, M., Bischl, B., & Richter, J. (2025). *mlr3tuning: Hyperparameter Optimization for ‘mlr3’*. Zenodo. <https://doi.org/10.5281/zenodo.15852621>
- Marc Becker, Jakob Richter, Michel Lang, Lennart Schneider, Bernd Bischl, Patrick Schratz, mb706, Sebastian Fischer, Michael Chirico, Manuel López-Ibáñez, Lionel Henry, & Lukas Burk. (2026). *mlr-org/bbotk: bbotk 1.13.1* [Computer software]. Zenodo. <https://doi.org/10.5281/ZENODO.20669808>
- Becker, M., Schneider, L., Binder, M., Kotthoff, L., & Bischl, B. (2026). *mlr3mbo: Bayesian Optimization in R*. arXiv. <https://doi.org/10.48550/ARXIV.2603.29730>
- Bengtsson, H. (2021). A Unifying Framework for Parallel and Distributed Processing in R using Futures. *The R Journal*, 13(2), 208. <https://doi.org/10.32614/rj-2021-048>
- Bergstra, J., Yamins, D., & Cox, D. (2013). Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. *Proceedings of the 30th International Conference on Machine Learning*, 115–123. <https://proceedings.mlr.press/v28/bergstra13.html>
- Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., Thomas, J., Ullmann, T., Becker, M., Boulesteix, A., Deng, D., & Lindauer, M. (2023). Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *WIREs Data Mining and Knowledge Discovery*, 13(2). <https://doi.org/10.1002/widm.1484>
- Bischl, B., Casalicchio, G., Das, T., Feurer, M., Fischer, S., Gijsbers, P., Mukherjee, S., Müller, A. C., Németh, L., Oala, L., Purucker, L., Ravi, S., van Rijn, J. N., Singh, P., Vanschoren, J., van der Velde, J., & Wever, M. (2025). OpenML: Insights from 10 years and more than a thousand papers. *Patterns*, 6(7), 101317. <https://doi.org/10.1016/j.patter.2025.101317>
- Bouthillier, X., Corneau-Tremblay, F., Notoraptor, Tsirigotis, C., Setepenre, Dong, L., Bronzi, M., Reyhane Askari, Dendi Suhubdy, Normandin, F., Abergeron, Simnol22, Breuleux, O., Wagner, S., Satyaog, Alain, G., Bertrand, H., Michael, Serdyuk, D., ... Lamblin, P. (2023). *Epistimio/orion: New Multi-Task Warm-starting and Dashboard Prototype* [Computer software]. Zenodo. <https://doi.org/10.5281/ZENODO.3478592>
- Carnell, R. (2026). *lhs: Latin hypercube samples* [Manual]. <https://doi.org/10.32614/CRAN.package.lhs>
- Egele, R., Balaprakash, P., Wiggins, G. M., & Eifert, B. (2025). DeepHyper: A Python Package for Massively Parallel Hyperparameter Optimization in Machine Learning. *Journal of Open Source Software*, 10(109), 7975. <https://doi.org/10.21105/joss.07975>
- Egelé, R., Guyon, I., Vishwanath, V., & Balaprakash, P. (2023). Asynchronous Decentralized Bayesian Optimization for Large Scale Hyperparameter Optimization. *2023 IEEE 19th International Conference on e-Science (e-Science)*, 1–10. <https://doi.org/10.1109/e-science58273.2023.10254839>

- Fischer, S., Zobolas, J., Sonabend, R., Becker, M., Lang, M., Binder, M., Schneider, L., Burk, L., Schratz, P., Jaeger, B. C., Lauer, S. A., Kapsner, L. A., Mücke, M., Wang, Z., Pulatov, D., Ganz, K., Funk, H., Harutyunyan, L., Camilleri, P., ... Bischl, B. (2025). *mlr3extralearners: Expanding the mlr3 Ecosystem with Community-Driven Learner Integration*. *Journal of Open Source Software*, 10(115), 8331. <https://doi.org/10.21105/joss.08331>
- FitzJohn, R., & Ashton, R. (2026). *rrq: Simple Redis queue* [Manual]. <https://github.com/mrc-ide/rrq>
- Fleck, S. (2026). *lgr: a fully featured logging framework* [Manual]. <https://doi.org/10.32614/CRAN.package.lgr>
- Gao, C. (2025). *mirai: Minimalist Async Evaluation Framework for R*. Zenodo. <https://doi.org/10.5281/zenodo.7912722>
- Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 80–112. <https://doi.org/10.1145/2363.2433>
- Ginsbourger, D., Le Riche, R., & Carraro, L. (2010). Kriging Is Well-Suited to Parallelize Optimization. In *Adaptation Learning and Optimization* (pp. 131–162). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-10701-6_6
- Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., & Sculley, D. (2017). Google Vizier: A Service for Black-Box Optimization. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1487–1495. <https://doi.org/10.1145/3097983.3098043>
- Head, T., MechCoder, Louppe, G., Iaroslav Shcherbatyi, Fcharras, Zé Vinícius, Cmmalone, Schröder, C., Nel215, Campos, N., Young, T., Cereda, S., Fan, T., Rene-Rex, Kejia (KJ) Shi, Schwabedal, J., Carlosdanielcsantos, Hvass-Labs, Pak, M., ... Fabisch, A. (2018). *scikit-optimize/scikit-optimize: v0.5.2* [Computer software]. Zenodo. <https://doi.org/10.5281/ZENODO.1207017>
- Kandasamy, K., Vysyaraju, K. R., Neiswanger, W., Paria, B., Collins, C. R., Schneider, J., Poczos, B., & Xing, E. P. (2020). Tuning Hyperparameters without Grad Students: Scalable and Robust Bayesian Optimisation with Dragonfly. *Journal of Machine Learning Research*, 21(81), 1–27. <http://www.jmlr.org/papers/v21/18-223.html>
- Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T.-Y. (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Advances in Neural Information Processing Systems*, 30, 3146–3156. https://papers.nips.cc/paper_files/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html
- Landau, W. M. (2026). *crew: a distributed worker launcher framework* [Manual]. <https://wlandau.github.io/crew/>
- Lang, M., Binder, M., Richter, J., Schratz, P., Pfisterer, F., Coors, S., Au, Q., Casalicchio, G., Kotthoff, L., & Bischl, B. (2019). *mlr3: A modern object-oriented machine learning framework in R*. *Journal of Open Source Software*, 4(44), 1903. <https://doi.org/10.21105/joss.01903>
- Lang, M., Bischl, B., & Surmann, D. (2017). batchtools: Tools for R to work on batch systems. *The Journal of Open Source Software*, 2(10), 135. <https://doi.org/10.21105/joss.00135>
- Lang, M., & Fischer, S. (2026). *mlr3oml: Connector Between mlr3 and OpenML* [Manual]. <https://doi.org/10.32614/CRAN.package.ml3oml>
- Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Ben-tzur, J., Hardt, M., Recht, B., & Talwalkar, A. (2020). A System for Massively Parallel Hyperparameter Tuning. *Proceedings of Machine Learning and Systems*, 2, 230–246. https://proceedings.mlsys.org/paper_files/paper/2020/hash/a06f20b349c6cf09a6b171c71b88bbfc-Abstract.html
- Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., & Stoica, I. (2018). *Tune: A Research Platform for Distributed Model Selection and Training*. arXiv. <https://doi.org/10.48550/ARXIV.1807.05118>
- Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., & Hutter, F. (2022). SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research*, 23(54), 1–9. <http://jmlr.org/papers/v23/21-0888.html>
- Mersmann, O., Beleites, C., Hurling, R., Friedman, A., & Ulrich, J. M. (2024). *microbenchmark: Accurate Timing Functions*. <https://doi.org/10.32614/CRAN.package.microbenchmark>
- Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., & Stoica, I. (2018). Ray: A Distributed Framework for Emerging AI Applications. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, 561–577. <https://www.usenix.org/conference/osdi18/presentation/moritz>
- Nii, H. P. (1986). The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2), 38–53. <https://doi.org/10.1609/aimag.v7i2.537>
- O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., & Invernizzi, L. (2019). *Keras Tuner*. <https://github.com/keras-team/keras-tuner>
- Olson, M., Santorella, E., Tiao, L. C., Cakmak, S., Garrard, M., Daulton, S., Lin, Z. J., Ament, S., Beckerman, B., Onofrey, E., Igusti, P., Lara, C., Letham, B., Cardoso, C., Shen, S. S., Lin, A. C., Grange, M., Kashtelyan, E., Eriksson, D., ... Bakshy, E. (2025). Ax: A Platform for Adaptive Experimentation. *Proceedings of the Fourth*

International Conference on Automated Machine Learning, 21/1–25. <https://proceedings.mlr.press/v293/olson25a.html>

R Special Interest Group on Databases (R-SIG-DB), Wickham, H., & Müller, K. (2024). *DBI: R database interface* [Manual]. <https://dbi.r-dbi.org>

Rocklin, M. (2015). Dask: Parallel Computation with Blocked algorithms and Task Scheduling. *Proceedings of the Python in Science Conference*, 126–132. <https://doi.org/10.25080/majora-7b98e3ed-013>

Rossini, A. J., Tierney, L., & Li, N. (2007). Simple Parallel Statistical Computing in R. *Journal of Computational and Graphical Statistics*, 16(2), 399–420. <https://doi.org/10.1198/106186007x178979>

Salinas, D., Seeger, M., Klein, A., Perrone, V., Wistuba, M., & Archambeau, C. (2022). Syne Tune: A Library for Large Scale Hyperparameter Tuning and Reproducible Research. *Proceedings of the First International Conference on Automated Machine Learning*, 16/1–23. <https://proceedings.mlr.press/v188/salinas22a.html>

Wickham, H. (2011). testthat: Get Started with Testing. *The R Journal*, 3(1), 5. <https://doi.org/10.32614/rj-2011-002>

Wickham, H. (2016). ggplot2. In *Use R!* Springer International Publishing. <https://doi.org/10.1007/978-3-319-24277-4>

Wickham, H., Pedersen, T. L., & Seidel, D. (2025). *scales: Scale functions for visualization* [Manual]. <https://doi.org/10.32614/CRAN.package.scales>

Wilke, C. O. (2025). *cowplot: Streamlined plot theme and plot annotations for 'ggplot2'* [Manual]. <https://wilkelab.org/cowplot/>

Wright, M. N., & Ziegler, A. (2017). ranger : A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. *Journal of Statistical Software*, 77(1). <https://doi.org/10.18637/jss.v077.i01>

Appendix

Table 3 Runtime of `$fetch_finished_tasks()` with and without caching as a function of the number of tasks, number of parameters or results, and payload size. When caching is enabled, the cache holds all previously fetched tasks except the most recent one, so that each call retrieves only a single new result from the database. The runtime is reported as the median with the 95th percentile in parentheses over 1,000 replicates for up to 1,000 tasks, 100 replicates for 10,000 tasks, and 10 replicates for 100,000 tasks.

Number of Tasks	Number of Parameters / Results	Payload Size	Runtime (ms)	Runtime Cache (ms)
1	1	1	1.08 (3.19)	1.11 (3.26)
1	1	10	1.14 (3.27)	1.07 (3.28)
1	1	100	1.13 (3.12)	1.07 (3.00)
1	10	1	1.13 (3.32)	1.10 (3.29)
1	10	10	1.09 (2.84)	1.19 (3.05)
1	10	100	1.10 (3.37)	1.15 (2.65)
1	100	1	1.19 (3.59)	1.28 (3.18)
1	100	10	1.20 (3.52)	1.23 (3.63)
1	100	100	1.39 (3.78)	1.33 (3.86)
10	1	1	1.41 (3.74)	1.09 (3.24)
10	1	10	1.43 (3.81)	1.10 (3.33)
10	1	100	1.44 (3.49)	1.12 (3.24)
10	10	1	1.43 (3.86)	1.09 (3.30)
10	10	10	1.53 (2.52)	1.10 (2.87)
10	10	100	1.52 (1.95)	1.11 (3.39)
10	100	1	1.82 (3.31)	1.23 (3.66)
10	100	10	1.98 (3.69)	1.28 (1.50)
10	100	100	2.47 (7.73)	1.30 (2.52)
100	1	1	4.11 (9.54)	1.15 (2.49)
100	1	10	4.44 (11.13)	1.15 (3.22)
100	1	100	4.40 (9.63)	1.16 (3.08)
100	10	1	4.55 (9.44)	1.20 (2.52)

Number of Tasks	Number of Parameters / Results	Payload Size	Runtime (ms)	Runtime Cache (ms)
100	10	10	4.90 (10.67)	1.19 (3.19)
100	10	100	5.48 (11.28)	1.23 (3.33)
100	100	1	8.26 (14.24)	1.59 (4.17)
100	100	10	9.35 (17.08)	1.50 (4.03)
100	100	100	17.47 (137.19)	1.60 (4.20)
1,000	1	1	38.93 (57.40)	1.26 (3.72)
1,000	1	10	37.59 (49.12)	1.29 (3.22)
1,000	1	100	38.57 (57.29)	1.29 (3.47)
1,000	10	1	38.57 (50.38)	1.47 (3.21)
1,000	10	10	43.13 (57.46)	1.43 (3.93)
1,000	10	100	49.16 (184.47)	1.49 (3.88)
1,000	100	1	82.07 (91.55)	3.24 (5.53)
1,000	100	10	100.34 (148.00)	3.25 (6.13)
1,000	100	100	308.26 (360.19)	2.78 (6.33)
10,000	1	1	305.06 (351.18)	1.60 (4.23)
10,000	1	10	466.67 (524.38)	1.67 (4.42)
10,000	1	100	331.99 (507.94)	1.74 (4.53)
10,000	10	1	555.92 (581.49)	3.15 (6.68)
10,000	10	10	436.40 (673.95)	3.80 (9.14)
10,000	10	100	825.37 (1,015.53)	8.42 (16.69)
10,000	100	1	1,413.02 (1,564.17)	24.88 (486.43)
10,000	100	10	1,377.38 (1,870.65)	25.06 (96.38)
10,000	100	100	2,204.29 (2,528.53)	24.93 (748.25)
100,000	1	1	3,667.42 (3,997.82)	12.70 (37.58)
100,000	1	10	3,662.08 (4,210.95)	6.72 (27.55)
100,000	1	100	4,120.23 (4,480.65)	20.10 (77.98)
100,000	10	1	4,331.54 (4,750.86)	27.12 (62.05)
100,000	10	10	5,497.61 (5,826.04)	40.10 (72.11)
100,000	10	100	6,611.68 (7,147.28)	41.13 (64.47)
100,000	100	1	18,637.08 (22,791.05)	575.03 (7,251.52)
100,000	100	10	22,156.63 (26,338.94)	294.34 (2,407.24)
100,000	100	100	25,561.06 (29,777.61)	335.69 (422.81)

Table 4 Hyperparameters of LightGBM optimized in the benchmark experiments. The table shows the lower and upper bounds of the search space for each hyperparameter, along with any special properties such as logscale transformations.

Hyperparameter	Lower	Upper	Properties
learning_rate	1e-3	1	Logscale
feature_fraction	0.1	1	
min_data_in_leaf	1	200	
num_leaves	10	255	
extra_trees	-	-	Logical value
lambda_l1	1e-3	1e3	Logscale
lambda_l2	1e-3	1e3	Logscale
min_gain_to_split	1e-3	0.1	Logscale
num_iterations	1	5,000	Internal tuning parameter

Table 5 Datasets used in the benchmark experiments. The table shows the OpenML (Bischl et al., 2025) task ID, name, number of instances, number of features, and number of classes for each dataset.

Task ID	Name	Instances	Features	Classes
31	credit-g	1,000	20	2
3945	KDDCup09_appetency	50,000	230	2
7592	adult	48,842	14	2
189354	airlines	539,383	7	2

Table 6 Runtime breakdown of the BO algorithms across the four benchmark tasks. For each task–algorithm combination, the table reports the cumulative runtime of the learners, the surrogate model, and the optimizer in seconds, the wall-clock time in seconds, the learner utilization in percent, and the final classification error. The learners, surrogate, and optimizer columns are cumulative runtimes summed across all workers. All entries are the mean over five replications with the standard deviation in parentheses.

Task	Algorithm	Learners (s)	Surrogate (s)	Optimizer (s)	Wall- clock (s)	Learner Utilization (%)	Classification Error
credit-g	CL	326 (22)	201 (13)	331 (18)	575 (25)	0.1 (0.0)	0.1918 (0.0030)
	ACBO	287 (14)	235 (4)	356 (5)	593 (2)	0.1 (0.0)	0.1910 (0.0028)
	ADBO	11,164 (325)	211,614 (413)	29,172 (340)	594 (1)	4.2 (0.1)	0.1838 (0.0016)
KDDCup09 appetency	CL	3,493 (358)	23 (1)	74 (1)	725 (2)	1.1 (0.1)	0.0175 (0.0000)
	ACBO	12,408 (521)	239 (5)	348 (5)	588 (2)	4.7 (0.2)	0.0175 (0.0000)
	ADBO	201,426 (2,170)	37,155 (1,676)	9,165 (221)	590 (0)	76.2 (0.8)	0.0174 (0.0000)
adult	CL	4,271 (523)	123 (10)	242 (14)	580 (36)	1.6 (0.1)	0.1227 (0.0001)
	ACBO	5,809 (1,486)	230 (7)	362 (7)	592 (2)	2.2 (0.6)	0.1226 (0.0003)
	ADBO	149,507 (2,067)	91,394 (2,003)	13,774 (300)	594 (0)	56.2 (0.8)	0.1222 (0.0001)
airlines	CL	27,396 (7,143)	19 (4)	61 (12)	664 (103)	9.1 (1.6)	0.3236 (0.0004)
	ACBO	83,108 (20,212)	220 (21)	364 (18)	585 (4)	31.7 (7.6)	0.3235 (0.0003)
	ADBO	260,517 (929)	293 (348)	567 (421)	586 (1)	99.2 (0.5)	0.3233 (0.0001)

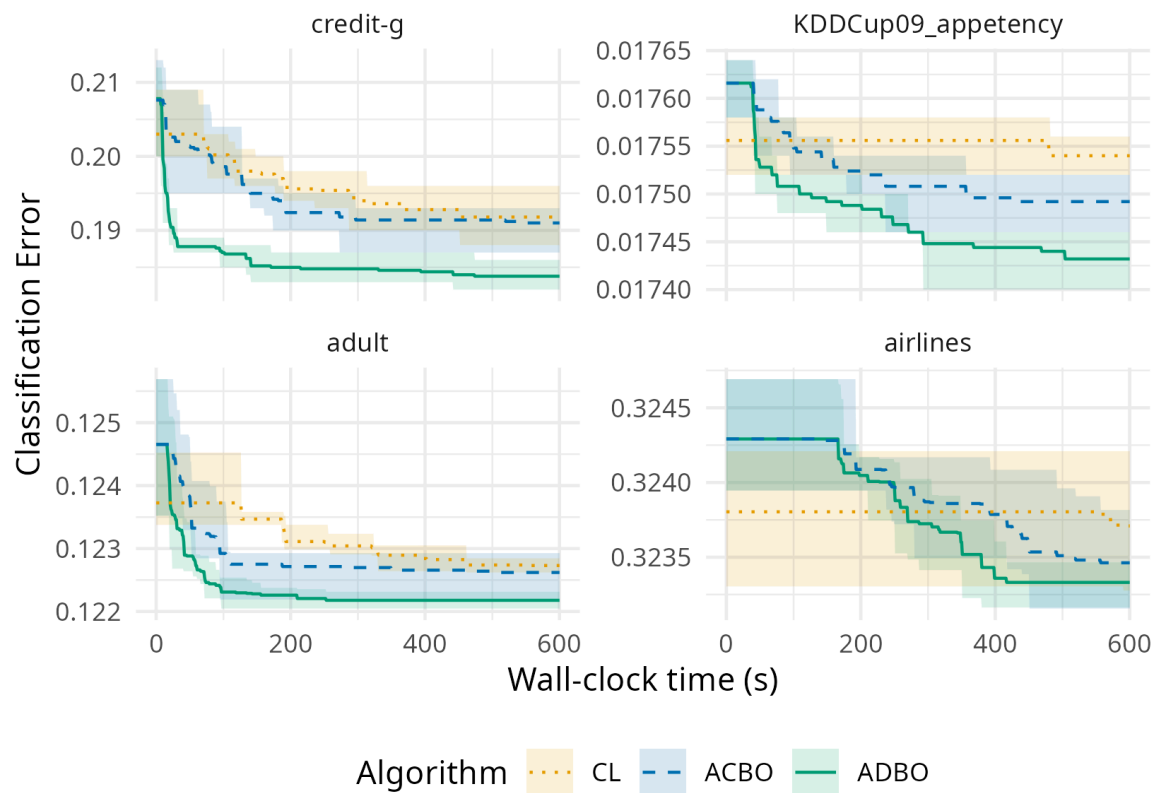


Figure 5 Incumbent classification error over wall-clock time of CL (dotted, orange), ACBO (dashed, blue), and ADBO (solid, green). Lines show the mean incumbent over the five replications. The shaded bands go from the best to the worst replication. All algorithms start from the same initial design of 100 configurations, evaluated before the timed runs (Section 5.1). Each trace begins with the first completed evaluation proposed by the BO loop. For CL this is the completion of the first batch of 44 configurations. ACBO and ADBO update the incumbent after every completed evaluation.

COLOPHON Typeset with Typst 0.15.0 from the authors' Quarto source. Template jrepstat.typ at commit 65ddfd0. Set in STIX Two Text, STIX Two Math and JetBrains Mono. Built 19 October 2026.