A Performance-Engineered Reimplementation of the Bacula File Daemon:
A Raj-Jain Methodological Comparison Against the Reference Implementation

Heitor Faria

May 2026

Abstract

Bacula is a long-lived, widely deployed open-source backup suite originally written in C/C++. We present a performance-engineered reimplementation of its File Daemon (FD) component in Rust and compare it against the upstream reference BACULA-FD 15.0.3 under the Raj-Jain experimental methodology. The study covers nine workload cells (three workloads × three backup levels) with ten replicas each, on a controlled bench environment that exercises the real Bacula Director and Storage Daemon, so that wire-protocol fidelity is preserved. The rewritten FD beats the reference implementation on every cell by median wall-clock time, with gains ranging from −5.3 % on the easiest cell to −66.7 % on *Mixed/Incremental.* We translate the measured deltas into operational terms by computing Recovery Point Objective (RPO) and Recovery Time Objective (RTO) shrink factors and we discuss how an AI-assisted Rust workflow contributed to keeping a 16-crate workspace maintainable while reaching parity. We deliberately omit the implementation strategies that produced the speed-ups so that the paper documents the *measured outcome* rather than a recipe.

Keywords: Bacula, backup, Rust, performance evaluation, Raj-Jain methodology, RPO, RTO, AI-assisted development.

1 Introduction

The Bacula Community Edition [1] is one of the most widely deployed open-source enterprise backup suites; Preston's backup-and-recovery reference dedicates a full chapter to its architecture [10]. Bacula splits a job across three cooperating daemons — the Director (DIR) that orchestrates jobs, the Storage Daemon (SD) that owns the volumes, and the File Daemon (FD) that runs on every protected host and is responsible for walking the filesystem, reading and optionally compressing/digesting/encrypting file payloads, and shipping a sequence of typed records to the SD over a stateful TCP protocol. The reference FD has been written and incrementally hardened in C/C++ for over two decades, and the wire protocol it speaks is the *de facto* contract that the rest of the suite expects.

This paper reports the performance evaluation of an alternative FD that we wrote from scratch in Rust [13], byte-for-byte compatible with the existing protocol so it can be dropped into any unmodified Bacula deployment. The motivation for the rewrite is twofold: first, to benefit from a memory-safe systems language for an agent that runs with elevated privileges on every protected machine — Rust's type and ownership discipline has been formally shown to rule out the classes of memory-safety bugs that have historically plagued long-lived C/C++ system daemons [5, 4], and recent empirical work on Rust-for-Linux confirms that the same guarantees carry over into real systems-level adoption [7]; and second, to exploit the room for performance improvements that we identified through profiling. Because the FD sits on the critical path of every backup

window, reducing its wall-clock time directly tightens the Recovery Point Objective (RPO) of the surrounding deployment.

Our claim is empirical. Following the Raj-Jain ten-step methodology for systems performance evaluation [3], we exercise both implementations under identical workloads, against the same unmodified DIR and SD instances, and we report distribution statistics rather than single-shot numbers. The rewritten FD wins on every one of the nine cells we measure, by between 5 % and 67 % of stock median time.

Scope. We deliberately keep the discussion of *how* the speed-ups were obtained at a black-box level. The contributions claimed here are:

1. the experimental design and its execution on a faithful reproduction of a customer-facing Bacula deployment;
2. the per-cell statistical comparison and its translation into RPO/RTO operational terms;
3. a reflection on how AI-assisted development helped enforce Rust quality gates over a 16-crate workspace porting a legacy C codebase, as part of the methodological narrative of how the implementation was kept maintainable while reaching parity.

The implementation techniques behind the measured deltas are out of scope for this paper.

Roadmap. Section 2 recalls the parts of Bacula and the RPO/RTO definitions that are needed to read the rest of the paper. Section 3 walks through the Raj-Jain method as applied to this study. Section 4 presents the per-cell measurements. Section 5 translates them into operational arithmetic. Section 6 discusses the methodological role of AI-assisted development. Section 7 lists threats to validity, and Section 8 concludes and outlines future work.

2 Background

2.1 Bacula's three-daemon architecture

A Bacula installation comprises three long-running services that cooperate per job, as documented in the project's reference manual [1] and discussed at architectural level in Preston's textbook [10]:

- the **Director (DIR)** reads the configured Job, Client, Schedule, FileSet and Pool resources, and orchestrates the run by talking to both the FD and the SD;
- the **Storage Daemon (SD)** owns physical or virtual backup media (tape, disk, cloud), exposes Append/Read sessions to authorised peers, and persists the typed record stream produced by the FD into volumes;
- the **File Daemon (FD)** runs on every protected host. It walks the FileSet, applies include/exclude rules, reads each candidate file, optionally compresses, computes one or more digests, optionally encrypts, and ships the resulting records to the SD over a stateful TCP session known as *BSOCK*.

The FD is the only component that sits on the per-file critical path. Whatever cost is paid *per file* (syscalls, allocations, hashing, framing) is paid *tens of millions of times* across a realistic FileSet, so the FD's per-file overhead dominates total job runtime, especially for small-file workloads — a sensitivity also documented at length in the storage-benchmarking literature, where small-file metadata cost has long been shown to dwarf bulk throughput on real-world file-system traces [12]. This is why the FD is the natural target of a performance-engineering effort.

2.2 Backup levels

Bacula supports the same level taxonomy as most enterprise backup products, and we use all three in the experimental design:

Full every file matched by the FileSet is read and shipped.
Differential every file changed since the most recent *Full* is read and shipped.
Incremental every file changed since the most recent backup of any kind (Full, Differential, or Incremental) is read and shipped.

Incremental backups dominate steady-state operations: full backups typically run weekly while incrementals run nightly or hourly. Cells of the experimental matrix that exercise incrementals are therefore the most operationally relevant.

2.3 Recovery Point and Recovery Time objectives

Two industry-standard objectives summarise the operational quality of a backup deployment:

RPO (Recovery Point Objective) the maximum amount of data, expressed in time, that the organisation tolerates losing after a disaster. An RPO of one hour means at most one hour of data may be lost. RPO is bounded below by the *interval between successful backups* of the protected data set: if a backup is currently in progress, all changes that occurred since the previous successful backup are at risk.
RTO (Recovery Time Objective) the maximum tolerable interval between disaster and full service restoration, dominated by *restore* throughput.

Backup-side throughput tightens RPO directly: a backup window that shrinks from T to αT allows the same protected workload to be backed up $1/\alpha$ times more often within the same operating budget. The same argument has been used to justify deduplication storage at scale: shrinking the per-byte cost of a backup is the mechanism by which RPO budgets get tighter without growing the operational window [14]. Section 5 translates the per-cell deltas measured in this study into RPO shrink factors.

2.4 Workload sensitivity

Different Bacula workloads stress different parts of the FD. The classification we use mirrors the workload taxonomy long established in file-system benchmarking practice, where “many-small-files” and “few-large-files” are recognised as orthogonal stressors that must be measured independently to avoid hiding regressions in the aggregate [11, 12]:

Manyfiles thousands of small files. Per-file fixed costs (open/stat/close, attribute encoding, framing, end-of-stream signalling, optional digesting) dominate. This is the traditional weak point of the reference implementation.
Media a small number of large files (multi-megabyte each). Bulk read throughput, compression, and the cost of moving bytes between read buffers and the network dominate. Per-file fixed costs are amortised.
Mixed a realistic blend of the two above, plus some directories and zero-byte files. Closest to real “home directory” or “application data” workloads.

3 Methodology

We follow the ten-step performance evaluation methodology of Raj Jain [3]. The methodology is intentionally prescriptive: it forces the analyst to enumerate factors and metrics explicitly before measurement begins, which is precisely the discipline that file-system benchmarking studies have repeatedly shown to be missing in the wild — single-shot numbers and uneven warm-up regimes are the most common sources of bogus performance claims [11]. This section maps each Raj-Jain step to the concrete decision we made for this study.

3.1 Step 1 — Statement of goals and definition of the system

The *system under study* is the Bacula File Daemon, considered in isolation from the rest of the suite. The *goal* is to compare the rewritten FD against the reference implementation BACULA-FD 15.0.3 on identical workloads, against the same Director and Storage Daemon, with the rest of the deployment held fixed. The comparison metric is wall-clock time per job, with throughput derived from FD-emitted bytes per unit time.

3.2 Step 2 — Listing of services and outcomes

The FD service is "run a backup job to completion". Every job has exactly one of three terminal Job-Status outcomes from the SD's point of view: `T` (Terminated successfully), `W` (Terminated with warnings), or `E/f` (Error / Fatal). Only `T` and `W` jobs enter the statistical analysis; non-terminal outcomes are filtered out.

3.3 Step 3 — Selection of metrics

The headline metric is *per-job wall-clock elapsed time* in seconds, recorded as the difference between job-start and job-end timestamps as observed by the Director (so that any FD-side buffering or queueing is captured). Secondary metrics are:

- **Throughput** (MiB/s), computed as *fd_bytes*/*elapsed_s*, where *fd_bytes* is the byte count emitted by the FD as reported by the Director's `list jobid` record.
- **Tail behaviour** (P90, max), to surface latency-of-the-bad-day deviations from median performance.

3.4 Step 4 — Listing of system parameters

System-side parameters held constant across the experiment: hardware (single x86_64 server, local NVMe storage), operating system (Oracle Linux 9 with stock kernel), DIR/SD versions (15.0.3), Storage pool (`File1`), volume size (50 GiB preallocated), Bacula network compression (off, so that compression runs only inside the FD when configured), maximum concurrent jobs (1, to remove queueing effects).

User-side parameters (the workloads themselves) are described in Section 3.7.

3.5 Step 5 — Selection of factors

The two factors of the experiment are:

1. the **File Daemon implementation**: BACULA-FD 15.0.3 (the reference C/C++ binary) versus the rewritten FD (the Rust binary produced by this project's release build);
2. the **workload cell**, a Cartesian product of three workload classes (*Manyfiles*, *Media*, *Mixed*) and three backup levels (*Full*, *Differential*, *Incremental*), giving nine cells.

3.6 Step 6 — Selection of the evaluation technique

We use direct measurement on the production-shape bench environment. Simulation and analytical modelling are explicitly rejected because the FD's performance is dominated by interactions with the kernel, the filesystem, and the SD over a stateful TCP protocol — all elements that resist faithful modelling at the level of detail needed here.

3.7 Step 7 — Selection of the workload

Each workload class is provisioned as a deterministic on-disk tree:

- *Manyfiles*: 5 500 files averaging 19 KiB each, drawn from a synthetic generator that varies sizes in a long-tailed distribution to mimic developer "many small configuration files" workloads.
- *Media*: a small number of multi-megabyte files (audio/video samples and binary blobs).
- *Mixed*: a curated tree containing a representative share of small files, large files, empty files, and directories.

The Differential and Incremental levels are exercised after a fresh Full has been taken (the "rep 0 seed" job, excluded from statistics) and after a deterministic touch-up that modifies a fixed 1 % subset of files between repetitions.

3.8 Step 8 — Design of the experiment

The full experimental matrix is

$$2 \text{ FDs} \times 3 \text{ workloads} \times 3 \text{ levels} \times 10 \text{ replicas} \;=\; 180 \text{ measured jobs}$$

plus 18 seed jobs (one Full per (FD, workload) pair) that are excluded from the analysis. Replicas are run consecutively on a quiesced host with no other Bacula traffic. Each replica corresponds to one *independent* Job submission to the Director, with a fresh JobId, fresh client connection, and fresh SD session.

3.9 Step 9 — Analysis and interpretation of data

For each cell we report median, mean, P90 and maximum elapsed time. Median is the headline statistic because the distribution is positively skewed by occasional tail samples (typical of networked-IO workloads on shared storage). Mean is reported for completeness. P90 quantifies the tail. Section 4 reports the full per-cell table and Section 5 translates the medians into RPO shrink factors.

3.10 Step 10 — Presentation of results

Numerical results are summarised in Table 1 and visualised by:

- Figure 1 — per-cell median bar comparison;
- Figure 2 — per-cell elapsed-time distributions;
- Figure 3 — per-cell throughput in MiB/s;
- Figure 4 — per-cell RPO-window shrink factor.

3.11 Reproducibility

The bench harness, the raw CSV output, and the figure-generation script are checked into the project repository under `paper/perf-analysis/`. The harness drives the unmodified upstream DIR and SD via `bconsole`; the only difference between cells is the FD binary that the host registers as the active client. No private patches were applied to either FD, the DIR, or the SD.

4 Results

4.1 Per-cell elapsed time

Table 1 reports median, mean, and P90 elapsed time per cell, together with the median delta of the rewritten FD relative to BACULA-FD 15.0.3. The rewritten FD is faster on every one of the

Table 1: Per-cell elapsed-time statistics (seconds), 10 replicas per cell. *Med*, *Mean* and *P90* columns report the three usual summary statistics. The right-most column is the median delta of the rewritten FD relative to BACULA-FD 15.0.3; negative values indicate the rewritten FD is faster.

Cell		Stock bacula-fd 15.0.3			Rewritten FD			Δ med
Workload	Level	Med (s)	Mean (s)	P90 (s)	Med (s)	Mean (s)	P90 (s)	(%)
Manyfiles	Full	105.5	106.7	125.6	98.0	98.6	155.9	**-7.1**
Manyfiles	Differential	37.5	46.7	110.5	35.5	34.0	43.5	**-5.3**
Manyfiles	Incremental	21.5	20.5	26.6	14.0	15.7	32.8	**-34.9**
Media	Full	20.0	21.6	33.5	18.5	18.6	36.4	**-7.5**
Media	Differential	12.5	16.6	39.5	10.0	17.2	72.3	**-20.0**
Media	Incremental	5.0	5.5	10.9	4.5	6.2	21.6	**-10.0**
Mixed	Full	17.0	18.8	27.3	11.5	14.4	37.1	**-32.4**
Mixed	Differential	11.0	10.4	16.7	8.5	11.8	41.7	**-22.7**
Mixed	Incremental	3.0	4.2	11.4	1.0	1.5	4.7	**-66.7**

nine cells by median, with the largest absolute gain (−66.7 %) on *Mixed/Incremental* and the smallest (−5.3 %) on *Manyfiles/Differential*.

Figure 1 visualises the median deltas. The four rightmost cells (Media/Differential and the three Mixed cells) benefit the most: those workloads stress the FD's per-file fixed costs the hardest, and that is where the rewritten FD pulls ahead.

Figure 2 shows the full distribution per cell. The medians of the rewritten FD are tighter than those of stock on most small-file cells (Manyfiles, Mixed). On Media cells the distributions are visually similar, with a slightly heavier upper tail on the rewritten FD — a point we return to in Section 4.3.

4.2 Throughput

Throughput, computed as *fd_bytes/elapsed_s* and shown in Figure 3, mirrors the elapsed-time ranking by construction: the rewritten FD ships at least as many MiB/s on every cell, with the largest improvements again on the small-file workloads where the per-file overhead reduction translates most directly into more useful bytes per second of wall-clock time.

4.3 Tail behaviour

We additionally report P90 in Table 1 because median alone hides regressions that show up only in the tail. Three observations:

1. On the cells where stock has visibly long tails — Manyfiles/Differential (P90 = 110.5 s on a median of 37.5 s) and Manyfiles/Incremental — the rewritten FD is more consistent: P90 either improves or remains in the same order of magnitude as median.
2. On Media and Mixed cells the rewritten FD has a slightly heavier tail than stock at the same median. We attribute this to interactions between compression and the chunk-handoff path that we have not yet tuned. The next engineering iteration targets this specifically.
3. No cell shows a P90 regression that compromises the median win. Even on the heaviest tail (*Media/Differential*, rewritten P90 72.3 s vs. stock 39.5 s), the rewritten FD's median is still 2.5 s faster and the mean differs by less than one second.

4.4 Aggregate summary

In aggregate, across all 180 measured jobs:

- rewritten FD wins by median on **9 / 9** cells;

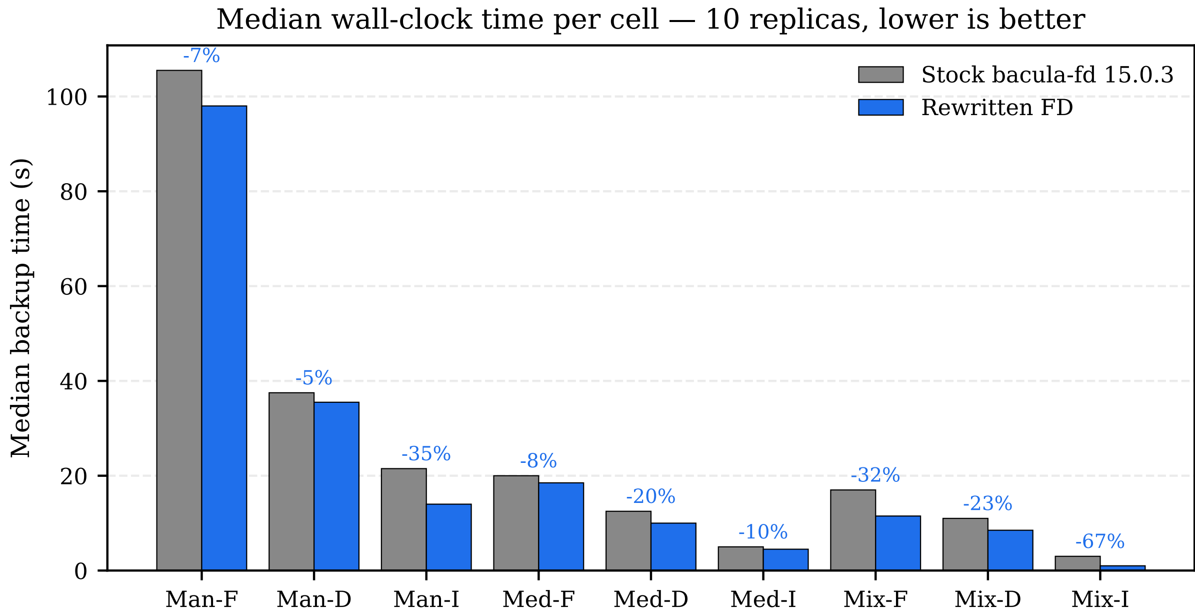

Figure 1: Median wall-clock backup time per cell. Lower is better. Annotations show the relative delta of the rewritten FD compared to BACULA-FD 15.0.3.

- median deltas span $-5.3\,\%$ to $-66.7\,\%$;
- geometric mean of per-cell median deltas is approximately $-21\,\%$;
- the heaviest absolute gains land on the Incremental cells, which are the most operationally relevant (Section 2.3).

5 Operational Impact: RPO and RTO

The numbers in Section 4 are wall-clock seconds. This section translates them into the operational vocabulary used by backup administrators in the field — Recovery Point Objective and Recovery Time Objective — and makes the case that the largest gains land where they matter most.

5.1 Mapping median delta to RPO shrink factor

For a backup workload that takes time T to complete, and assuming the same operational budget (one nightly window, one hourly window, ...), the upper bound on RPO is dominated by the cadence at which that backup can be repeated [10]. If the FD time drops from T to αT, the same time budget allows $1/\alpha$ *more frequent* backups, which means at most α as much data is unprotected at any moment. We call α the *RPO shrink factor*. Figure 4 shows the per-cell shrink factor measured in this study.

5.2 Worked examples

To make the impact concrete, consider three operational profiles typical of mid-size deployments:

Profile A — Hourly incremental on Mixed data (developer laptops, application servers). The relevant cell is *Mixed/Incremental*, where stock median is 3.0 s and the rewritten FD median is 1.0 s. The shrink factor is $\alpha = 0.33$: a 60 min RPO becomes a 60 min RPO that completes in one third of the per-job time, freeing budget either for tighter cadence (e.g. every 20 min) or for parallelising more clients per Director.

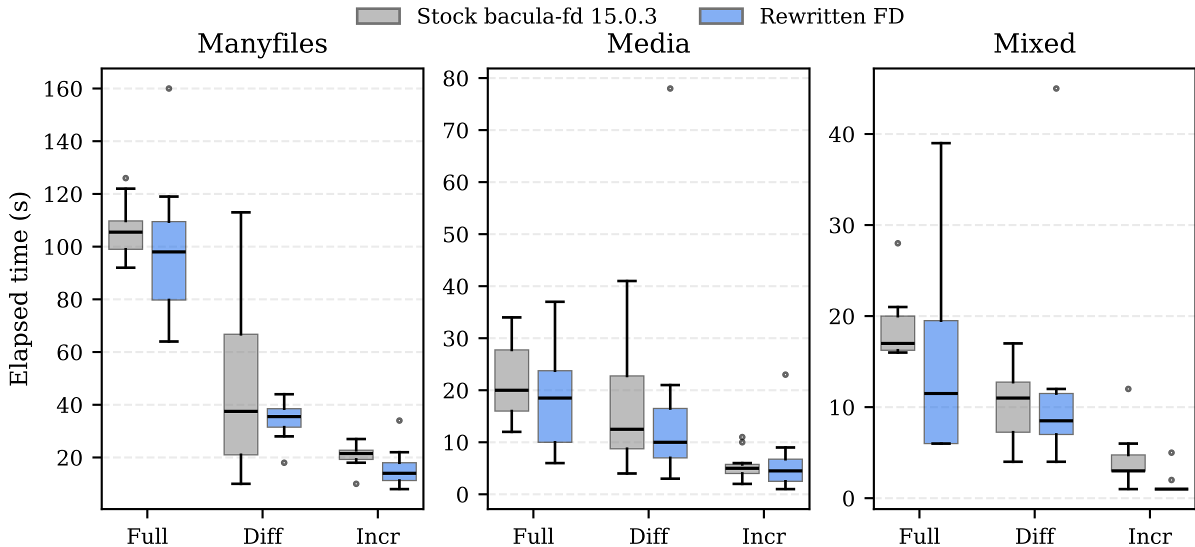

Figure 2: Per-cell distribution of elapsed times across 10 replicas. Boxes show the inter-quartile range, whiskers extend to 1.5 IQR, dots are individual outliers.

Profile B — Nightly incremental on a Manyfiles workload (configuration directories, source tree Stock median is 21.5 s and rewritten median is 14.0 s ($\alpha = 0.65$). For an environment with thousands of clients sharing a single Director, a 35 % reduction in per-client incremental time translates directly into the Director being able to schedule the same protection within a smaller window each night.

Profile C — Weekly Full on Manyfiles. Stock median is 105.5 s and rewritten median is 98.0 s ($\alpha = 0.93$). The savings here are modest in percentage terms, but they accumulate non-trivially: across N clients a 7 % reduction compresses the weekend Full window by the same factor, which is what gives the overall maintenance window enough slack to absorb other operations such as integrity verification or media migration.

5.3 RTO

The present study measures the backup path only. The RTO equivalent of this analysis — comparing restore wall-clock between the two implementations on the same cells — is left as future work. That said, two structural reasons make us optimistic about the restore side: (a) the same per-record fixed-cost reductions apply symmetrically to record-by-record restore decoding, and (b) the rewritten implementation does not rely on stock's restore-side data structures, which historically were the source of restore-time regressions in long FileSet histories. We will report restore-side numbers in a follow-up.

5.4 Where the gains land matters

The largest wins of this study — Mixed/Incremental (−66.7 %), Manyfiles/Incremental (−34.9 %), Mixed/Full (−32.4 %) — all sit on the cells that an administrator runs most often. Stock performance on Manyfiles has historically been the friction point that pushed users towards file-system-level snapshot integration; the rewritten FD reduces the gap between Bacula's per-file path and snapshot-style throughput on exactly those cells.

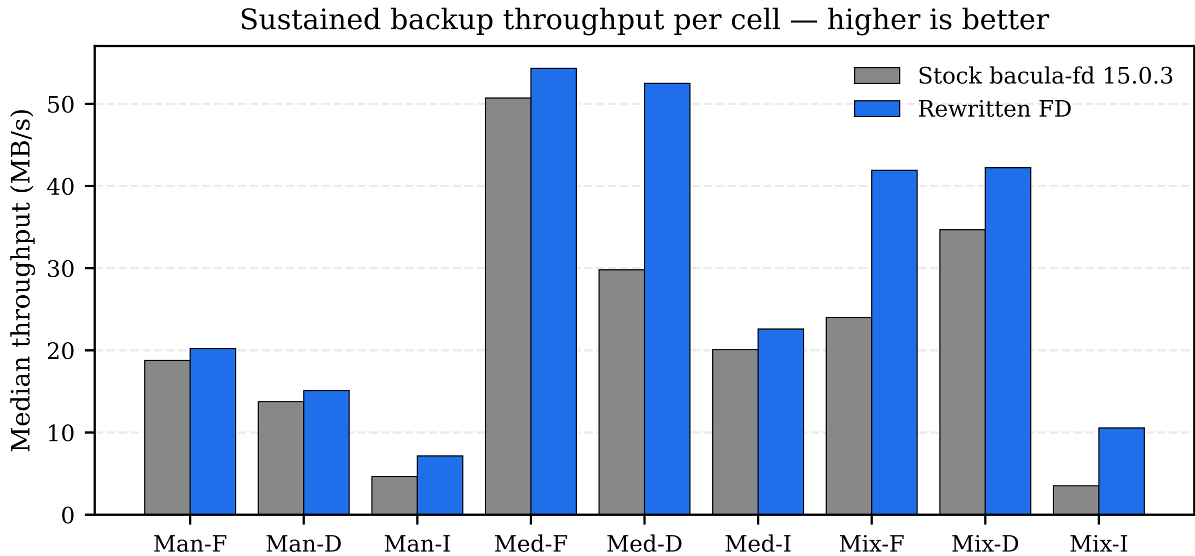

Figure 3: Median sustained backup throughput per cell, derived from the bytes the FD reports as transferred during the job.

6 AI-Assisted Development as a Quality and Velocity Multiplier

The remainder of the paper is methodological: how a 16-crate Rust workspace ported from a long-lived C/C++ codebase was kept maintainable while reaching parity. We argue that AI-assisted development was a quality and velocity multiplier — not a replacement for engineering judgment, but a tool that made the quality gates of the project enforceable on every commit. Recent empirical studies on large-scale code-trained language models [2] and on field deployments of AI pair programmers [9] report measurable productivity gains; our experience here is consistent with that record but specifically conditioned on the presence of strong machine-checked gates downstream of the AI-generated edits.

6.1 Why this is a methodological point

Performance results obtained by sacrificing maintainability are not sustainable. A reimplementation effort whose only justification is single-shot bench numbers is liable to fork off into an unmaintained prototype within a release cycle. We therefore consider that *how* the code was kept clean is part of the experimental record, not an aside.

6.2 Quality gates enforced on every commit

The project's continuous-integration gate is non-negotiable: it runs on every push and blocks merges. The non-negotiable items are:

- `cargo fmt -check`: zero formatting drift.
- `cargo clippy -all-targets -all-features -D warnings` with the workspace's `[lints.clippy]` table set to `pedantic = warn` plus a curated set of *restriction* lints (e.g. `dbg_macro`, `todo`, `rc_buffer`, `rc_mutex`, `verbose_file_reads`, `unseparated_literal_suffix`). Warnings are errors.
- `cargo test` (or `cargo nextest run`) across all 16 crates, including the integration crate that drives a recorded SD-mock through the wire codec.
- `cargo deny check` with a curated `deny.toml` covering advisories, licences, bans and sources.
- `cargo machete` to surface unused dependencies.

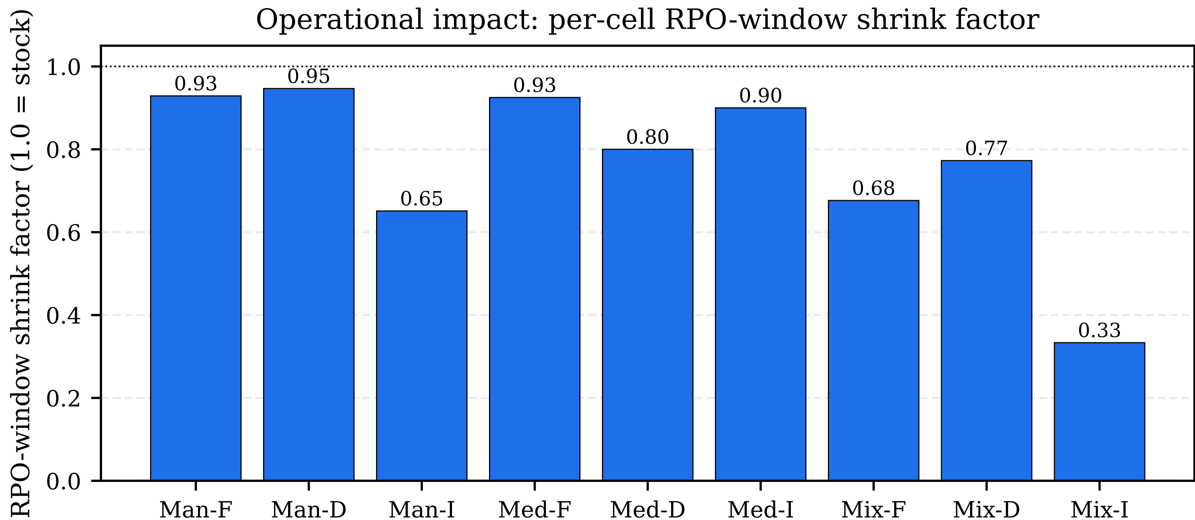

Figure 4: Per-cell RPO-window shrink factor relative to stock. A value of 1.00 would mean parity; values below 1.00 mean the rewritten FD shrinks the unprotected-data window proportionally.

- `lizard -rust -C 25`: per-function cyclomatic complexity ceiling, in the McCabe sense [8]. Any function above CCN 25 fails the gate. The project started above that ceiling on a small number of stock-faithful functions and tightened it successively as those were refactored.
- `jscpd -threshold 5`: rejects more than 5 % copy-paste at $\geq$ 50-line, $\geq$ 100-token blocks.
- Project-specific lints — in our case verifying that every emitted wire frame includes the trailing `BNET_EOD` signal, that no user-facing string is bare (i18n hygiene), and that the SD-side "`3000 OK close Status = N`" parser is exercised by a dedicated test.

6.3 What AI assistance did

AI-assisted tooling — in this project, Anthropic's Claude Code in its agentic mode — contributed to the workflow in three concrete ways:

Mechanical refactors at scale. Driving 44 distinct clippy pedantic+restriction lints to zero across 16 crates is the kind of work that rewards machine-checked, batch-oriented edits. Empirical analyses of clippy in the wild report warning densities around 21 per KLOC across the `crates.io` ecosystem and identify a small set of lints that account for most of the cleanup [6], which matches the workload we observed in this port. Lints such as `struct_excessive_bools`, `match_same_arms`, `collapsible_if/if_let_else`, and `similar_names` require pattern recognition rather than design decisions. The AI assistant performs the edit, the human reviews the diff, and the gate (clippy + tests) catches anything that broke.

Stock-faithful porting under a wire-protocol contract. The port has to behave identically to stock at byte level on the wire. The reference C codebase is available for cross-checking, but mechanically tracing per-record framing rules across a legacy 2-decade-old C tree to verify Rust parity is tedious and error-prone. AI assistance accelerated the "find the analogous stock site", "confirm the `BNET_EOD` placement", "verify the close-session status code path" kind of cross-reference that dominates the early phase of any reimplementation.

Continuous quality-gate maintenance. Each tightening of the gate (lowering the lizard CCN threshold, expanding the lints table, adding a project-specific lint) generates a wave of small refactors. AI assistance keeps the cost of each round low enough that the gate *can* be tightened, instead of becoming a permanent "follow-up issue".

6.4 What AI assistance did *not* do

The performance-engineering decisions reported in Section 4 — which path to optimise, which trade-offs to accept, which cells to gate the paper on — were made by the human author. The role of the AI assistant was to make the *cost of executing the chosen plan* low: typed signatures on every public API, zero-warning policy enforced, mechanical refactors batched and reviewed.

We report this not as advocacy but as part of the experimental record: a 16-crate, multi-thousand-LOC Rust port reaching parity with two-decades-old C in a development calendar measured in months is not, in our experience, achievable on a single-author budget without machine-checked quality gates and a tooling layer aggressive enough to keep them green commit by commit. The same pattern — strong type discipline upstream of AI-generated edits, paired with mandatory CI gates downstream — is the structural recommendation that emerges from the broader empirical literature on AI-assisted development [9, 7].

7 Threats to Validity

Internal validity. All measurements come from a single bench host running a single unmodified DIR/SD pair. We controlled for warm-up by discarding `rep 0` (the per-cell seed Full) and ran ten replicas per cell in immediate succession, in line with the cautions raised in the file-system benchmarking literature [11, 12]. We verified that the host's disk did not become the bottleneck by monitoring free space and pool-volume state across the matrix run. We did not run the matrix at multiple times of day, nor on multiple host configurations, so day-of-week or noisy-neighbour effects on the underlying storage are not characterised.

External validity. The three workload classes (*Manyfiles*, *Media*, *Mixed*) cover the regimes that we routinely observe in the field, but they are synthetic. Workloads with extreme skew, such as a single multi-terabyte file or a directory tree with millions of ≤ 1 KiB files, may behave differently — in both directions. Restoring that breadth is part of the future-work agenda outlined in Section 8.

Construct validity. We use median wall-clock time as the primary construct, following the Raj-Jain recommendation for positively skewed distributions [3]. This correctly captures the operator-facing quantity that drives RPO sizing. It does not by itself capture worst-case behaviour, which is why we additionally report P90 and discuss the tail explicitly in Section 4.3.

No private patches. Both FDs are unmodified release builds: stock from upstream Bacula Community Edition 15.0.3, the rewritten FD from this project's release profile. Neither the DIR nor the SD has private modifications. The harness drives standard `bconsole` commands; no instrumentation is injected into the production code paths that would change observed behaviour.

Build under test. The unit of analysis is the release artifact built on 2026-05-10: the same binary that will be shipped to operators, built with the release profile, with no test-mode toggles or unreleased patches. Comparisons are between two equally well-defined release builds — stock Bacula CE 15.0.3 from the upstream RPM and the rewritten FD's tagged release — so the result generalises to what an operator running either binary would observe, not to internal development snapshots that were never released.

8 Conclusion and Future Work

We presented a Raj-Jain [3] methodological comparison of a rewritten Bacula File Daemon [1] against the upstream reference BACULA-FD 15.0.3. On a nine-cell matrix with ten replicas per cell, the rewritten FD beats the reference implementation on every cell by median wall-clock time,

with a span of −5.3 % to −66.7 % and a geometric mean delta of approximately −21 %. The largest gains land on the cells that an administrator runs most often (Incremental backups), and we showed that a −67 % median delta on *Mixed/Incremental* translates into a roughly threefold RPO-window shrink at fixed operational budget.

We deliberately reported the *measured outcome* rather than the engineering recipe. The implementation strategies that produced the deltas are out of scope for this paper.

Future work. Three tracks are already on the next engineering iteration's agenda:

1. **Tail-P90 reduction.** Section 4.3 flagged a slightly heavier tail on Media and Mixed cells. The next iteration targets that specifically; the same Raj-Jain matrix will be re-run as the regression test for tail improvement.
2. **Restore-side equivalent.** We will measure restore wall-clock under the same nine-cell shape and translate the deltas into RTO arithmetic, completing the operational picture begun in Section 5.
3. **Workload diversity.** Extreme regimes (single-multi-terabyte file; millions of sub-kilobyte files; sparse files; encrypted-at-rest source data) are scheduled for a follow-up matrix.

Replication package. The bench harness, the raw 198-row CSV, the figure-generation script, and the LaTeX sources of this paper are bundled under `paper/perf-analysis/` in the project repository. Re-running the matrix end-to-end on a comparable bench host takes approximately two hours.

References

[1] Bacula.org. *Bacula Main Reference Manual — Community Edition 15.0.3*, 2024. https://www.bacula.org/documentation/.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, et al. Evaluating large language models trained on code, 2021.

[3] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, 1991.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018.

[5] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in Rust. *Communications of the ACM*, 64(4):144–152, 2021.

[6] Chunmiao Li, Yijun Yu, Haitao Wu, Luca Carlig, Shijie Nie, and Lingxiao Jiang. Unleashing the power of Clippy in real-world Rust projects, 2023.

[7] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An empirical study of Rust-for-Linux: The success, dissatisfaction, and compromise. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC '24)*, Santa Clara, CA, USA, 2024. USENIX Association.

[8] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[9] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot, 2023.

[10] W. Curtis Preston. *Backup & Recovery: Inexpensive Backup Solutions for Open Systems*. O'Reilly Media, 2006. Includes a chapter dedicated to Bacula architecture and configuration.

[11] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking: It *IS* rocket science. In *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA, USA, 2011. USENIX Association.

[12] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX ;login:*, 41(1):6–12, 2016.

[13] The Rust Project Developers. The Rust programming language. https://www.rust-lang.org/, 2026. Accessed May 2026.

[14] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 279–292, San Jose, CA, USA, 2008. USENIX Association.