ChatGPT

# Accelerating RCC Holotree Environment Performance

**Holotree Content-Addressed Caching:** Robocorp's *Holotree* system treats execution environments as content-addressed file stores. Instead of rebuilding full environments from scratch for each run or user, Holotree stores each unique file **only once** in a central library (the *hololib*), and uses a manifest (*catalog*) to assemble environments on demand [1] [2]. This means if two environments share Python X but differ in a few packages, those common files aren't duplicated – only the differing files are added. As the Robocorp docs note, "if one robot needs Python X and another needs version Y, usually only a few files need to change, so there is no sense in storing them multiple times" [2]. This powerful deduplication underpins many performance optimizations. Below, we explore high-leverage improvements across various scenarios, along with their performance implications and trade-offs.

## 1. Local Environment Restore & Rebuild Times

When running a robot locally (`rcc run` or `rcc ht restore`), RCC uses the Holotree cache to avoid redundant installs. Each environment is identified by a *blueprint* hash of its `conda.yaml` (environment spec) [3]. If the blueprint was built before, RCC can skip full resolution and use the cached result. However, initial builds or changes can still be slow due to environment solving and file operations. Key improvement areas include:

- **Faster Dependency Resolution:** Environment solving (conda/micromamba and pip) is often a bottleneck. Using **Mamba** (already integrated via Micromamba) is a big step, but further speedups are possible. For example, the new Pixi package manager (built on Rust's `rattler` library) resolves and installs in parallel, showing ~3× faster installs than Micromamba (and 10× vs Conda) in benchmarks [4]. Adopting similar strategies – parallel downloads, async I/O, and using lockfiles – can dramatically cut environment solve time. Ensuring **lockfile** support (so repeated builds skip solving) and possibly integrating Pixi or `rattler` for Python/PyPI dependency resolution would improve performance for complex envs (Pixi even provides a native pip dependency resolver to avoid pip's slow backtracking [5]).

- **Caching & Reuse of Packages:** RCC already caches environment files in hololib, but we can also cache at the package level. Ensure that Conda's package cache and pip's wheel cache are reused across builds. If the same version of a package is needed in a new env, Micromamba should fetch it from local cache instead of redownloading. In practice, holotree's design means if an environment is similar to a previous one, most files are already in hololib – only new package files are added, reducing build I/O. This acts as an implicit delta: *only new unique files (content not seen before) are written on environment creation* [1]. We should verify that *micromamba* is configured to use a persistent cache directory, so that even the initial extraction can be skipped for already-seen packages.

- **Concurrent Installation Steps:** Look for opportunities to parallelize steps in the environment build. For example, conda package extraction and pip installations could be run concurrently if they don't depend on each other. Micromamba is fast at solving, but installation could leverage multiple CPU cores (extracting multiple packages in parallel). Also, any pip installs (if listed in `conda.yaml`) might be sped up by parallel pip operations or pre-built wheels. Minimizing pip's work (e.g. using constraints to avoid heavy backtracking) also helps – in one case, a pip resolution issue took *4+ hours to fail* due to backtracking through many versions [6]. Introducing a max-resolution limit or using locked requirements can "fail fast" in such scenarios, saving time [7] [6].

- **Incremental Environment Updates:** For iterative development, rebuilding an environment from scratch each time is overkill if only one or two dependencies change. A possible enhancement is an **incremental restore**: detect which new packages were added or removed compared to a previous blueprint and only resolve those differences. Since holotree catalogs list every file in an env, RCC could diff two catalogs and avoid re-copying files that remain identical. In practice, because hololib already contains identical files, the cost is mainly in preparing the new catalog. Still, avoiding even the bookkeeping for unchanged files could speed up the *"rcc ht restore"* for similar environments. Another idea is reusing an existing environment space if it's already *pristine* and matches the spec – i.e. skip the teardown/restore if nothing has been "dirtied." RCC by default resets environments to pristine state for reliability, but perhaps a checksum-based check could verify pristine-ness and skip a redundant restore step.

- **Filesystem Throughput & Concurrency:** Local restore involves creating a new environment folder (Holotree *space*) by copying or linking files from hololib. On SSDs this is I/O intensive but can be parallelized. RCC could traverse the hololib catalog with multiple threads to copy files concurrently. Care must be taken on HDDs (to avoid thrashing) and on CPU usage (computing hashes etc.), but a tuned thread pool can significantly cut wall-clock time on multi-core machines. Also consider the filesystem performance: Windows NTFS can be slow with massive numbers of small file operations. The RCC *speedtest* recommendation is to ensure the disk yields a positive score [8]. Using faster disks (NVMe) or tuning the OS (e.g. disabling real-time virus scanning on the cache directories) can improve restore times, and documentation can guide users on these best practices.

- **Pre-Resolved Environments (No-Build Mode):** RCC already supports creating prebuilt environments. For example, it has a *"no build"* configuration where a pre-created environment is used without internet access [9]. Leverage this by distributing known-good environment definitions with resolved dependencies (conda-lock or `rcc export`). If RCC sees an environment ID that's known and prebuilt, it can bypass Micromamba entirely and just unpack the ready files from hololib. In essence, treat environment resolution as a one-time cost that can be amortized across runs and machines.

**Trade-off considerations:** Aggressively reusing environments or skipping resets can risk "dirty" environments if not careful. The default approach is conservative (always restore pristine state) to avoid hidden state issues [10] [11]. Any optimization that skips a rebuild must ensure equivalently clean state (e.g. by verifying no file deviated). Parallelization and caching generally improve speed, but complexity rises – e.g. lockfile management and concurrency bugs. Overall, focusing on solver efficiency and file copy optimizations yields big wins for local usage.

## 2. Remote Pull Workflows (RCCRemote) Performance

The `rccremote` service allows client machines to pull pre-built environments over the network instead of building locally. This dramatically accelerates setup: *"when clients ask for environments, rccremote only relays the missing files, not the whole environment. After the first 2–4 environments, the number of files downloaded decreases significantly, and the visible environment build time drops drastically."* [12] This content-deduplication over the network is akin to how Docker or Nix only transfer layers not already present. To further streamline remote pulls, consider:

- **Reducing Round Trips:** The current pull protocol involves two steps – (1) GET a listing of needed parts (`/parts/<catalog>`) and (2) POST that list to get a zip of the missing content [13] [14]. This ensures minimal download size, but it incurs two HTTP requests sequentially. We could combine these into a single request if the client sends its known hashes up front. For example, the client could POST a bloom filter or list of existing library fingerprints along with the environment request, and the server responds with only the delta. This would eliminate the listing round-trip. The trade-off is a larger single request (client must send its inventory of content hashes, which could be thousands of entries), but for fully up-to-date clients this could be negligible (server would respond "nothing needed" in one step). **HTTP/2** or HTTP/3 could also be leveraged to allow the listing and content transfer in parallel streams if combining isn't practical.

- **Smarter Catalog Checks:** If a client already has a given environment's catalog and all parts, `rcc pull` should no-op quickly. RCC can maintain a local index of which catalogs are complete. Before contacting the remote, it can check `HasBlueprint` in hololib (which already verifies if a catalog and all its files exist) [15] [16]. Only if that returns false should it call out to rccremote. This "pre-check" saves time and network calls when the env is already satisfied (essentially a cache hit). If not fully present, the current method is already checksum-based at the file level (fingerprint hash names), so it inherently diff's what's missing.

- **Batching and Compression:** The rccremote server currently packages missing files into a single zip for download [17] [18]. This is good for reducing request overhead and compressing data. We should ensure the compression level is tuned for speed vs size – e.g. possibly use *store* or low-compression for already-compressed files (like `.pyc` or `.dll`), to save CPU at both ends. If network bandwidth is the bottleneck, a higher compression may be worth the CPU cost; but if CPU or disk I/O is limiting, then lighter compression (or even sending a tarball uncompressed) could be faster overall. It might be beneficial to dynamically adjust: for large binary-heavy environments, don't compress much, but for text-heavy or many small files, do compress.

- **Parallel Transfers (if needed):** Because all missing parts are zipped into one file, we're essentially utilizing a single stream. This simplifies consistency (all-or-nothing) but on very high-latency networks a single TCP stream might underutilize bandwidth. An enhancement could be chunking the zip and downloading parts in parallel or using multiple connections. However, this adds complexity and is usually unnecessary on typical LAN/VPN speeds. The current approach is likely sufficient, as one zip can saturate bandwidth and ensures atomic integrity (the catalog and parts arrive together).

- **Server-Side Caching & Throughput:** On the server, ensure that frequently requested environment data is cached in memory or easily accessible. If many clients ask for the same environment around

the same time, the server could reuse the computed zip file. For instance, rccremote could pre-compose the full environment archive when a new catalog is imported (essentially caching "environment layer" archives). Then serving it to multiple clients is just a file transfer (and it could even use HTTP range requests or a CDN). Currently, rccremote likely creates the zip on-demand per request [19]. Caching them (perhaps with an LRU of most popular env zips) could trade some storage for CPU savings. This works best when clients typically need whole environments (first-time pulls). For incremental pulls (only a few files missing), on-demand zips are fine.

- **Security & Integrity Checks:** Performance and integrity go hand-in-hand. The pull process should verify that downloaded layers match their expected hashes (since file fingerprints double as content hashes). The RCC client does check the HTTP response status and uses a SHA-256 on the entire zip to name it [20] [21], but verifying each file against its fingerprint on import could catch corruption (perhaps this happens via `CatalogCheck` after import [16]). Ensuring these checks are efficient (e.g. streaming verify during unzip rather than re-hashing files afterward) avoids costly rework if a bad download slips through.

In summary, the rccremote protocol is already optimized for minimal data transfer by using deduplication. The main gains will come from cutting any unnecessary latency (handshakes) and optimizing compression. The benefit is very clear: after a few environments, clients see drastically reduced download times [12], which translates to faster robot startup. Trade-offs lie in complexity – adding things like parallel transfers or huge client manifests might not pay off unless dealing with extremely large environments or unusual network conditions.

## 3. CI/CD Environments with Shared Caches & Distribution

In continuous integration/delivery pipelines, robots may run in ephemeral containers or VMs that lack persistent state. Without optimization, each CI job might rebuild the same environment, incurring big delays. We can accelerate CI/CD scenarios by **sharing holotree caches across builds** or pre-distributing environments:

- **Persistent Hololib Volume:** If using self-hosted runners or persistent agents, configure a shared hololib directory that multiple builds can use. RCC's *Shared Holotree* mode (designed for multi-user machines) can be repurposed here: enable the shared holotree at a system location (e.g. `/opt/robocorp/ht` on Linux) with appropriate permissions [22]. In CI, all build jobs on the same machine (or same Kubernetes node, etc.) will then pull from a single cache of environment files. This means if job A builds environment X, job B (on the same host) can restore environment X almost instantly from cache with no internet. Similarly, any overlapping dependencies between different environments benefit from the common store. The trade-off is that one must manage cache growth and concurrency (RCC uses file locks to serialize writes to the holotree [23], which should be respected across jobs to avoid race conditions).

- **Pipeline Caching of Hololib:** Many CI systems (GitHub Actions, GitLab CI, etc.) allow caching directories between jobs or pipeline runs. Teams can cache the `~/.robocorp/hololib` directory (and `holotree` if needed) as an artifact keyed by the environment spec. For example, use the hash of `conda.yaml` as a cache key. The first job that builds a new environment will populate the cache; subsequent jobs restore the cache and skip rebuild. Because hololib holds potentially many envs, a

more refined approach is caching per environment: use `rcc holotree export` (see below) to create a compressed archive of a built environment, store that artifact, and have jobs import it if needed. This avoids carrying the entire cache (which can grow large over time) through every job.

- **Pre-Built Environment Distribution:** A highly effective pattern is to **build environments in CI once, then reuse everywhere**. This is the idea behind Robocorp's *Builder* and *Importer* robots [24] [25]. For example, a nightly CI job can run on an internet-connected runner to resolve and build all needed environments (for various robots), exporting each to a `hololib.zip`. These zips can be stored in a package registry or file server. Then any CI job (even in an isolated network or fresh container) can download the appropriate zip and do `rcc holotree import`. This imports the environment files into its hololib in one shot, ready to use. The import step is much faster than a full build since it's just extracting files – and as a bonus, these files have already been virus-scanned and approved upstream (an enterprise requirement in some cases [26]). By separating build from run, teams can ensure that CI jobs spend minimal time on environment prep, focusing on executing tests or deployments.

- **Compressed Distribution (hololib.zip in CI):** Using compressed archives of environments has pros and cons. On one hand, transferring a zip is simpler and often faster than pulling thousands of individual files via conda/pip. The archive can also be compressed to reduce network transfer. On the other hand, compression/decompression adds overhead. If CI nodes have CPU to spare and network is a bottleneck, compression is beneficial; if network is fast (e.g. same data center) and CPU is limited, it might be faster to use an uncompressed tar. We could make this configurable. Robocorp's hololib export already creates a zip by default [27] – in tests, we should measure if using *store-only zip* vs *deflate* yields faster results given typical environment content. Often, conda packages include many already-compressed files (binary libraries, etc.), so compression gains might be modest.

- **Shared Cloud Cache/Registry:** For distributed CI (where jobs run on many machines), a central artifact repository can act as the environment cache. This is analogous to language-specific caches (like a Maven or npm cache). An internal S3 bucket or Nexus server could store holotree content by fingerprint. For example, after building an env, upload the entire `~/.robocorp/hololib` (or just the new files) to S3 keyed by their hash. Other runners, on encountering a missing file, could fetch it from S3. This is basically implementing a lightweight, decentralized version of rccremote (discussed more in section 7). The advantage in CI is that you don't need a long-running `rccremote` server – you leverage existing infra. The challenge is complexity: you'd have to integrate RCC's cache with the CI caching mechanism or custom scripts.

**Performance impact:** With these techniques, CI/CD pipelines can get near instant environment setups after the first build. For instance, one case study using similar methods (conda/Pixi in CI) achieved significant reduction in pipeline time by reusing environment packages and shipping environments as artifacts [28] [29]. The main trade-off is management overhead – ensuring caches don't become stale or too large. It's important to periodically clean old environment files (RCC provides `rcc holotree cleanup` or similar) to keep the cache lean [30]. But this effort is well worth the hours of build time saved in active CI environments.

## 4. Air-Gapped Distribution with Hololib.zip

Air-gapped or highly restricted environments (with no internet access) pose a special challenge: robots still need their Python libraries. RCC addresses this with `hololib.zip` – an export of all files needed for an environment that can be physically transferred and imported offline [31] . To optimize this scenario:

- **Optimize Import/Export Speed:** Creating the hololib zip involves reading potentially tens of thousands of files and writing them into an archive. This process can be I/O heavy. One improvement is to **avoid re-compression** of already-compressed files inside the zip. If the export process detects certain file types (e.g. `.pyc`, `.dll`, `.so`, `.zip` within site-packages), it could store them in the zip without compression to save time. The RCC export implementation already marks duplicate files to only add them once [32] [33] – meaning if the same file content appears multiple times across catalogs, it will be archived only once to keep the zip smaller. We should also ensure export uses multiple threads (one thread reading files and another writing to zip) to maximize throughput, as modern disks can handle concurrent read/write better than strictly serial operations.

- **Footprint Reduction Techniques:** The size of hololib.zip can be large (environments can be gigabytes). We can borrow ideas from tools like **conda-pack** and **pixi-pack**. For example, conda-pack can exclude caches, bytecode files, or test data that aren't needed at runtime to shrink the archive. RCC could offer options to strip certain files from the holotree when exporting (with caution). Also, **dependency pruning** could be applied: if some packages in the environment were installed but not actually used by the robot, an advanced analysis could omit them to reduce size – though this is more complex and generally conda environments include only needed packages.

- **Platform-Specific Packs:** A single hololib.zip is OS- and architecture-specific (you cannot use a Windows environment on Linux, etc.). In air-gapped scenarios, you might need to ship separate zips for each platform. To ease this, ensure the build system can produce multi-platform exports. For instance, if using cross-platform building or separate builder machines, standardize the output so that each hololib.zip is clearly labeled with platform and easily importable on the matching target. RCC's blueprint hash already includes platform info in catalog names (e.g. `...v12.win-64` vs `...v12.linux-64` ) [34] , preventing cross-OS confusion.

- **Fast Import with Hardlinks:** When importing the zip on the target machine ( `rcc holotree import hololib.zip` ), RCC will unzip files into the local hololib. We can accelerate this by using hardlinks if the zip is unpacked on the same volume as the hololib. One approach: unzip the archive into a temporary directory and then move the files into hololib using `os.rename` (which is O(1) if on same filesystem) or link them. However, since the archive likely already contains the structure intended for hololib, RCC's import might just extract directly into the cache directory. Ensuring that the extraction can use system optimizations (like Linux's copy-on-write reflinks or Windows block cloning if available) would speed it up. For example, on Btrfs or APFS, a tool could duplicate files nearly instantly. There is an existing `hardlink` utility on Linux that replaces identical file copies with hardlinks to save space and time [35] – RCC could similarly post-process the holotree to link any duplicates that might have been introduced (though ideally, the export avoids duplicates in the first place by design).

- **Validation vs Speed:** In offline mode, it's critical that the imported environment is exactly correct, since there's no online fallback. RCC might perform a content check after import (verifying all expected hashes). This is good for integrity but can double-read every file (once to unzip, again to verify). If the zip is signed or its integrity is assured, we might choose to trust it and skip re-hashing every file on import to save time. A compromise is to verify a random sample or just the catalog's hash. The user could be given a flag: `--verify-import` for a full check, defaulting to on in secure contexts but off if speed is paramount and the source is trusted.

Real-world analogs like **pixi-pack** take a slightly different approach: instead of bundling the fully installed files, pixi-pack collects the package binaries (conda packages) and on unpack it installs them in place [36] [37]. This avoids shipping duplicate files if multiple envs share packages and leverages package managers to do final install offline. RCC's holotree zip is more direct – it contains the ready-to-use file tree – which is faster to activate (no installation step) but possibly larger if there's overlap between archives. In an enterprise, one could distribute one giant hololib.zip that contains *all* common packages needed for many environments and import that everywhere (essentially a "mega-cache"). Clients would then have an almost complete library and could create multiple env spaces from it without internet. The downside is size and perhaps including unused content. The optimal balance might be shipping a base runtime hololib (Python, common libs) and then smaller incremental zips for specific robots.

In summary, to optimize air-gapped deployments we aim to compress smarter and extract faster. The benefit is clear: previously impossible installations (no internet) become feasible, and the time to get an environment up is just the time to transfer and unzip, which is often minutes instead of hours. Care must be taken to maintain reliability (can we trust the environment bundle?) and to handle updates (you may need a process for distributing updated zips when conda.yaml changes).

## 5. Filesystem-Level Optimizations (Hardlinks, Symlinks, OverlayFS)

At the core of Holotree's performance is how it handles files on disk. Currently, RCC likely copies files from the hololib cache into each new environment *space* (or uses some linking – we saw hints of counting "links" and "duplicates" in the code [38] [39]). There are several techniques to minimize actual data copying:

- **Hardlinks:** A hardlink makes a directory entry to the same file data on disk, so creating a hardlink is extremely fast (no file content is duplicated) [40]. If RCC were to hardlink files from hololib into the environment directory, environment creation would be very fast and consume virtually no extra disk for duplicates. The code's statistics (counting `links` vs `duplicate` vs `dirty`) suggests that indeed when a file is already in hololib, they might hardlink it instead of copying [41]. Using hardlinks has a *big caveat*: if the environment tries to modify that file, it would actually modify the shared cached file (since hardlinks are the same inode). For *pristine* environments that aren't modified by the running process, this is fine. But if any process accidentally writes to a library file, it corrupts the cache for everyone. To mitigate this, RCC can mark all files as read-only when linking them. Programs typically don't rewrite library files, and if they try, it will fail, preserving cache integrity. (If truly necessary, the user could copy-on-write that particular file, but most cases don't require it.) On Linux and macOS, non-root processes won't be able to remove the read-only flag easily on their own, and on Windows, we could use ACLs to protect the hololib. With these safeguards, **hardlinking is a top choice** for speed and deduplication: it offers *O(number of files)* environment creation time with minimal I/O. One must ensure the holotree and spaces reside on the same filesystem/partition, as hardlinks can't span volumes.

- **Symbolic Links (Symlinks):** Symlinks are another option, but slightly less ideal for this use-case. A symlink in the environment pointing to the file in hololib would also avoid copying data. Programs reading the file follow the link transparently. However, if a program tries to open the file for writing, it will write to the target (hololib), just like a hardlink. The difference is one could potentially detect or break this link. But unlike hardlinks, modifying a symlink's target affects all who use it too. Symlinks also introduce some path differences (some tools might treat symlinked files differently, and relative symlinks need careful construction if used). In an earlier update, Robocorp noted that holotree "fully supports symbolic links inside created environments" [42], likely referring to not breaking symlinks that are part of packages themselves. Using symlinks *to the cache* should be possible, but given similar write risks as hardlinks (plus extra overhead of resolving links), they are generally not superior. One niche benefit: you could locate the hololib on a read-only network share and symlink to it from local envs – but hardlinks wouldn't work in that scenario. If read-only network storage of common layers is desired, symlinks could implement that (with the network filesystem ensuring no writes).

- **Copy-on-write (CoW) Filesystems:** Using advanced filesystems or overlays can combine the benefits of linking with isolation. For example, **OverlayFS** on Linux can create a unified view: hololib content as the lower (read-only) layer and an empty upper layer for changes. When a robot runs, any attempt to modify a file will cause a copy-up (the original in lower remains intact, the upper gets a private copy). This is exactly how Docker containers implement copy-on-write layers. We could create an overlay mount for each environment space instead of physically copying files. The performance benefits are huge for large environments – thousands of files appear instantly from the lower layer, and only modified or new files ever get written to disk. Once the robot run is done, the upper layer can be discarded to revert to pristine state. The downside: OverlayFS requires root privileges (though user namespaces might allow it in some setups), and it's Linux-only. Windows has no direct equivalent (Windows containers use a filter driver that's not trivial to employ for user processes), and macOS has APFS snapshots/overlays but not easily accessible for this use. Another approach is using **Btrfs or ZFS** snapshotting: one could keep the hololib as a subvolume and snapshot it for each environment. Btrfs has reflink copy which allows cheap CoW copies of files (e.g. `cp --reflink`). Tools like `hardlink` or `duperemove` can retroactively deduplicate files via CoW on Btrfs/ZFS [35]. If RCC detects the filesystem supports it, it could choose to clone files rather than copy, achieving the same effect as hardlink (distinct files that share data until written). Implementing this is more low-level (OS-specific system calls), so it's an advanced optimization.

- **Filesystem Structure and Metadata:** Aside from *how* files are copied, consider *how many* files and directories are involved. Conda environments contain tens of thousands of files (each package has its own directory in `pkgs` and then files in env). There is a cost to creating lots of small files. One technique seen in Nix is to store content in a content-addressed store as big pack files, but Holotree is already on the granular side (file-level dedup). A possible improvement is reducing filesystem metadata overhead by storing groups of small files as one physical file in the cache (like a pack). However, this complicates access and is rarely worth it unless dealing with millions of tiny files. More straightforward is to ensure the filesystem is formatted with settings for many files (e.g., turn off atime on Linux to avoid updating access times on each read). RCC could recommend or even attempt to set `noatime` on the holotree folder to boost read speed.

- **Concurrency & Locking:** At the FS level, enabling multiple operations in parallel can speed things up, but we have to avoid conflicts. Holotree uses a global lock when importing layers to avoid

concurrent writes [23]. For read/restore operations, it may allow concurrency since just reading cache and linking should be safe. If multiple envs are being created at once (e.g., parallel `rcc run` processes), using links means they mostly do metadata operations (which are fine to do concurrently). This should scale well across processes. Monitoring tools could be added to detect if I/O becomes a bottleneck (e.g., sequential copying saturating disk vs. parallel).

**Trade-off table:**

| Method | Benefit (Speed/Space) | Drawbacks / Considerations |
|---|---|---|
| **Hardlinks** | Instant file presence; no extra space for dupes.<br>Environment assembly is very fast (just create links). | Risk of cache corruption if written to.<br>Must ensure read-only usage. Same-volume only. |
| **Symlinks** | No data copy; can reside on different volume (cache on network drive, env local). | Writes follow link to cache (same risk as above).<br>Some tools might treat symlinks unexpectedly. Slight runtime overhead to resolve. |
| **OverlayFS / CoW** | Near-instant env creation; true copy-on-write isolation (writes don't touch cache).<br>Great for large env churn in Linux containers. | Linux-only (root needed). Not available on Windows/macOS directly.<br>More complex to set up mounts per env. |
| **Standard Copy** | Simple and universal; each env fully separate (no shared inode). | Slow for large envs (lots of I/O). Duplicates waste space (mitigated by holotree dedup of *cache* but not of env copy). |

In practice, a combination might be used: on Windows and Mac (where overlay isn't an option), use hardlinks with caution; on Linux for CI or power users, offer an optional flag to use OverlayFS for maximal performance. The default could remain copying (for safety and compatibility) but advanced users could opt in to linking. Given that holotree's philosophy is to avoid duplication, it's likely RCC already uses either hardlinks or copies+dedup internally. Leveraging OS features more aggressively could yield further speed-ups with minimal downsides in controlled scenarios.

## 6. UX and Protocol Improvements

Enhancing user experience (UX) and refining protocols can indirectly improve performance by reducing mistakes, retries, or downtime:

- **Smarter `rcc pull` UX:** When a user runs `rcc holotree pull <env>`, the tool could proactively inform them if the environment is already up-to-date or partially present. For instance: "Environment already cached, skipping download." If not, showing a progress bar for downloading parts (e.g. "Downloading 120/500 files (60% complete)") would help users understand the wait. Currently, because it bundles into one zip, progress might only be known at the byte level – but RCC could at least print the count of missing parts before downloading [43] [14]. This gives a sense of scope ("500 files (200MB) to download"). A more advanced idea: *pre-check layers* – if the remote has

an updated catalog for an environment the user already has, it might allow a "diff" mode to just pull differences. This overlaps with the earlier logic of checking blueprint existence, but extended: for example, if Control Room says environment X was updated (new catalog version), RCC could compare the old vs new and only fetch new files. This is essentially what it does by fingerprint, but making it explicit in UX ("updating environment to latest version...downloading 10 new files") would be helpful.

- **Checksum-Based Diffing & Validation:** We touched on ensuring integrity; from a UX perspective, if a pull fails (say network issue), RCC could detect partial content and resume. Because each file has a hash, it can skip already downloaded parts on retry. Implementing a resume might mean keeping the incomplete zip and requesting only the missing pieces again – not trivial with the current single-zip approach. Alternatively, if the zip download fails, RCC could fall back to requesting files individually (since it knows which were missing). A smarter protocol might be: rather than sending a big zip, allow chunked transfers or multiple range requests. However, this complicates the simple design. Perhaps a better approach is simply **robust error handling**: if the zip is corrupted or incomplete, the client should not delete already written files – it should only fetch the ones not in place. The code does check each file existence with the `/parts` listing first [43], so presumably if a previous attempt got some files into hololib, a second attempt will skip them.

- **User-Controlled Layer Management:** Exposing more commands to manage the holotree can empower users to optimize performance themselves. For instance, a command to *pre-fetch* or *pre-pull* certain common layers (like "download Python3.9 core environment to cache") could warm up caches outside of robot run time. Or a command to list which environments (catalogs) are present and their completeness status. This gives transparency – users can prune unused envs (freeing space and speeding up lookups) [44] or identify that they should pull an update.

- **Protocol Standardization (OCI/Conda index):** While the current HTTP API is custom, adopting a standard could improve compatibility and possibly performance via existing tooling. For example, treating the hololib content as an OCI registry means we could use container registries and their optimizations (CDNs, layer caching, etc.). Each environment's catalog could correspond to an OCI image manifest, and each hololib file (or group of files) to an OCI layer blob. `rcc pull` could then internally use an OCI client to fetch layers by digest, which gives resume support, content verification, and parallel downloads out-of-the-box. This crosses into the "alternative implementations" territory (discussed below), but from a UX perspective it means users could potentially use familiar tools (`docker pull` or `oras` CLI) to fetch robot environments, and RCC would just consume the local layers. Similarly, leveraging the **Conda package index** format: if we push prebuilt env files as a conda channel (with each environment as a metapackage depending on certain package versions), then standard conda clients could retrieve them. However, this is likely more complex than beneficial, so clarity and simplicity of RCC's own protocol might be preferable.

- **Logging and Diagnostics:** Performance issues often hide in the details. Providing users with tools like `rcc diagnostics` or verbose timing logs (which RCC does keep via `TimelineBegin/End` internally [45] [46]) can help pinpoint slow steps. For example, if environment resolution is slow, a warning could suggest using a `--strict` mode or check for conflicts. Or if file copy is slow, maybe it's due to disk or antivirus. A good UX is to detect common pain points (like pip taking > X minutes in resolution) and surface a hint ("It looks like pip is struggling; consider adding version constraints or

check network connectivity"). This doesn't speed up by itself, but it shortens the feedback loop for users to take action.

Overall, UX and protocol improvements ensure that the high-performance capabilities of RCC are used effectively. A robust, user-friendly pull process reduces failed runs and wasted time. And adopting proven protocols (where appropriate) can offload some performance work to battle-tested infrastructure (for example, OCI registries are highly optimized for large binary distribution). The trade-off usually comes down to adding these features without making RCC usage more complex – the goal is to keep it easy while exposing power features for those who need them.

## 7. Alternative Implementations & Decentralized Layer Sharing

Finally, it's worth exploring bigger-picture rearchitecting that could yield performance and scalability gains. This includes how the Control Room, remote layer server, or even the concept of a layer registry could be implemented differently:

- **OCI/Docker Image Registry for Environments:** As alluded above, one idea is to piggyback on the container ecosystem. An environment (holotree catalog + files) can be analogized to a Docker image: a set of layers (files) identified by hashes. Robocorp could provide an OCI-compatible registry for environment layers (or allow users to use their own). Instead of `rccremote` custom server, RCC could push layers to a registry (each file or package group as a layer TAR, with the content hash as digest). Clients (RCC) would then pull using standard protocols, which automatically skip layers already present. The **benefit:** OCI registries are ubiquitous and optimized – they handle large scale distribution, caching, geographic replication, and resume. Docker's pull mechanism already checks for existing layers by checksum and only downloads missing ones, much like rccremote does [12]. By mapping holotree's library to OCI layers, we essentially get an "environment registry." Real-world practice: some teams already containerize their robot environments – building a Docker image with the robot code and dependencies pre-installed – which sidesteps RCC entirely. But not everyone can use containers (especially for desktop UI automation). This approach gives a similar benefit (fast deployment) while running on host.

- **Go-Based Control Room Layer Service:** If not leveraging OCI, another path is to implement the **Control Room's environment handling in Go** (to match RCC's performance and type-safety). For example, the Control Room could incorporate a layer registry service written in Go that interfaces with cloud storage. Go's concurrency and low overhead would allow handling many client connections for environment downloads efficiently (much like how Nexus or Artifactory manages artifacts). It could also manage authentication, access control, and auditing of who downloaded what (important in enterprise). The current `rccremote` is already in Go and intended to be run on a user-provided VM [47]. Reimplementing similar logic as a cloud service could simplify setups (no need for users to host their own rccremote for sharing envs between machines). The challenge is ensuring security and multi-tenant isolation if hosted centrally.

- **S3 or Cloud Blob Backend:** We've mentioned storing layers on S3 a few times – this is a very practical decentralized strategy. Instead of a custom server, **use the cloud as the "dumb" storage** and let clients be smart. For instance, when `rcc holotree export` produces a zip, upload that to S3. Or even upload individual files: every hololib file could be an object at key `<hash>` in a bucket.

Then a simple service (or even a static JSON file) can serve as a catalog index mapping environment blueprint -> list of file hashes. Clients could retrieve this index (JSON or similar), then directly GET each needed file from S3 (using parallel requests). S3 will efficiently deliver files and handle range requests, etc. Many package managers implement caches like this (homebrew bottles, for example, are just blob storage downloads). The benefits are no custom server code and scalability; the downsides are lack of built-in diff-ing (the client has to handle what to get, which RCC already does). Also, transferring thousands of small files individually might be slower than one zip, but many HTTP libraries and S3 clients can parallelize and batch efficiently. A middle ground is to store *layer bundles* on S3 – e.g., combine frequently co-occurring files into one object. Designing the bundling is an interesting problem (similar to choosing Docker layer contents or Nix store nar bundles). Perhaps grouping by conda package or by directory could be logical.

- **IPFS or Peer-to-Peer Deduplication:** Decentralized networks like **IPFS** take content-addressing to the global scale. Since holotree file hashes are basically content IDs, one could imagine adding the hololib files to IPFS. Then any machine with those files could serve any other. For example, an office could have one machine that built environment X; another machine needing it could retrieve many pieces from its peer rather than a central server. IPFS automatically finds and downloads content by hash from any available nodes. This could reduce central server load and make use of local network speeds. It also provides dedup across the entire network – if two different envs share a file and any peer has it, others can get it. The overhead is running IPFS nodes and the complexity of ensuring data availability/pinning. It may be overkill for most users, but it's a forward-thinking design. In a closed environment, a simpler P2P could be implemented: e.g., each robot runner could have the ability to serve its hololib to others on the LAN (perhaps via a simple UDP discovery and HTTP server). This is complex to implement securely, but it's a thought for speeding up large-scale deployments (imagine hundreds of robots in a facility sharing layers amongst each other).

- **Comparison to Nix/Guix:** Nix's binary cache is conceptually similar to holotree (store objects identified by hash, downloaded on demand). Nix has a concept of *substituters* – basically URLs to binary caches. They serve `.nar` archives of store paths. A substituter could be an S3 bucket or an HTTP server. Nix only downloads if the store path hash matches the desired derivation (ensuring content match). Robocorp could emulate this by having a simple URL pattern for each catalog and file hash. In effect, RCC would try `https://cache.robocorp.com/hololib/<hash>` for each missing file. If present (and authorized), it downloads it; if not, falls back to building. This is very much like how Nix works and would make RCC more decentralized and resilient (you could have multiple caches in a list). Guix (another functional package manager) actually can use IPFS as a source for substitutes, showing it's viable.

- **Implementations in Other Languages:** The note about implementing in Go might also be hinting at re-writing some components currently in Python. For example, perhaps parts of the Control Room or environment preparation tasks are in Python and slower. A Go implementation could boost performance for those backend tasks. If the Control Room triggers environment builds, a Go builder might solve environments faster (again using rattler or micromamba libraries). This reduces the time between an environment being requested and it being available in the remote cache.

**Trade-offs:** Embracing container or package manager standards can greatly increase performance and interoperability, but might reduce fine-grained control. For instance, if we use Docker/OCI, we have to fit within its layer model (tar archives, typically per-image layering, which might not map perfectly to how

holotree splits files). Also, pushing large layers to a registry might introduce new bottlenecks (like needing to have a beefy registry or pay for egress). A custom lightweight solution (like the current rccremote) is tailored to only what's needed and nothing more. However, as usage grows, standing on the shoulders of proven distribution tech could avoid reinventing wheels – for example, Docker registries excel at exactly this kind of problem (content-addressed large file distribution), and companies often have infrastructures for them.

In a decentralized scenario (like P2P or IPFS), complexity is the major con. Ensuring content is available and not tampered with (IPFS would still need some access control overlay for enterprise) can be non-trivial. But conceptually, the more we can distribute the load (peers, mirrors, caches), the faster environment provisioning can be, especially at scale.

---

## Conclusion

Through these improvements, RCC's holotree-based system can achieve even faster and more streamlined performance across local development, remote execution, CI pipelines, and offline use cases. The core principle is to **avoid doing redundant work**: don't rebuild what's already built, don't re-download what's already on disk, and don't copy data unnecessarily. Holotree's content-addressable cache already provides a strong foundation for this by storing each unique file once and referencing it via catalogs [1]. Building on that with techniques like hardlinking, parallelism, smarter distribution protocols, and borrowing ideas from tools like Nix and Pixi can yield substantial gains:

- *Faster local runs* by caching and reusing dependency resolutions and linking files instead of copying.
- *Lightning-fast remote pulls* that transfer only deltas – as evidenced by drastic drop-off in downloaded data after a few environments [12].
- *Efficient CI/CD* through shared caches and prebuilt artifacts, cutting environment prep from minutes to seconds.
- *Offline deployments* made practical with optimized hololib bundles and quick imports.
- *Filesystems optimized* to minimize overhead (exploiting hardlinks, CoW, etc., where safe).
- *Enhanced protocols* that leverage content hashes for integrity and standardize on high-performance delivery mechanisms.
- *Scalable architecture* that can integrate with existing ecosystems (containers, cloud storage, P2P) for distributing environment layers.

Each improvement comes with trade-offs – complexity vs. speed, generality vs. platform specifics, upfront effort vs. runtime benefit – but the overall impact is a more robust and high-performing automation platform. By adopting these techniques, Robocorp's RCC can continue to scale to larger environments, more simultaneous robots, and stricter deployment scenarios without sacrificing speed or reliability. The end result for users is a smoother experience: environments that "just appear" when needed, with minimal wait time, whether on a developer's laptop or across an enterprise fleet of robots.

**Sources:**

- Robocorp RCC documentation and code, e.g. holotree design and shared cache feature [2] [1]
- RCC performance notes on rccremote (partial environment transfer) [12]
- Pixi and conda ecosystem insights (performance comparisons and packaging strategies) [4] [36]

- Pip resolver issue illustrating worst-case scenario to avoid [6]
- General filesystem optimization knowledge (hardlink vs copy-on-write) [35]

---


[1] [3] [10] [11] vocabulary.md
https://github.com/joshyorko/rcc/blob/119e0a485f6bd102162619ee6a6940b358a37dc0/docs/vocabulary.md

[2] [22] Enabling RCC Shared Holotree on multi-user machines | Robocorp documentation
https://sema4.ai/docs/automation/faq/shared-holotree

[4] [5] [28] 7 Reasons to Switch from Conda to Pixi | prefix.dev
https://prefix.dev/blog/pixi_a_fast_conda_alternative

[6] [7] Improving failures at `ResolutionTooDeep` to include more context · Issue #11480 · pypa/pip · GitHub
https://github.com/pypa/pip/issues/11480

[8] [12] [24] [25] [26] [47] Setup pre-built environments using RCC | Robocorp documentation
https://sema4.ai/docs/automation/rcc/pre-built-environments

[9] [27] [30] [31] [42] [44] history.md
https://github.com/joshyorko/rcc/blob/119e0a485f6bd102162619ee6a6940b358a37dc0/docs/history.md

[13] [14] [17] [18] [19] [20] [21] [23] [43] [45] [46] pull.go
https://github.com/joshyorko/rcc/blob/119e0a485f6bd102162619ee6a6940b358a37dc0/operations/pull.go

[15] [16] [32] [33] [34] [38] [39] [41] library.go
https://github.com/joshyorko/rcc/blob/119e0a485f6bd102162619ee6a6940b358a37dc0/htfs/library.go

[29] [36] [37] Shipping conda environments to production using pixi | QuantCo Engineering Blog
https://tech.quantco.com/blog/pixi-production

[35] hardlink(1) - Linux manual page - man7.org
https://man7.org/linux/man-pages/man1/hardlink.1.html

[40] Can a hard link break the file system structure?
https://unix.stackexchange.com/questions/180379/can-a-hard-link-break-the-file-system-structure