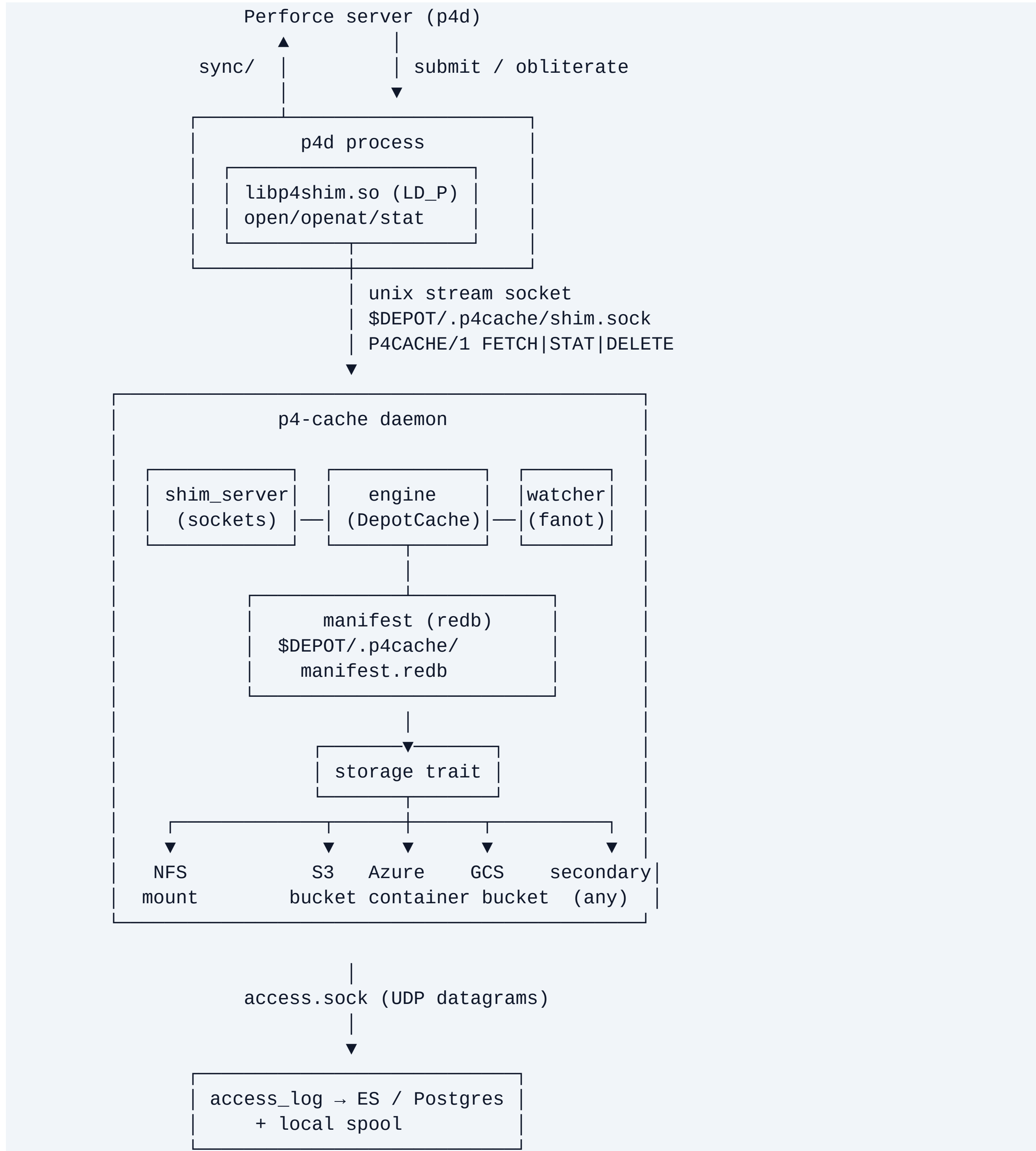
p4-cache — Architecture Deep-Dive

Reference document for enterprise architecture reviews. The artifact a storage architect, security architect, and Perforce admin team will walk through page-by-page in a 90-minute session.

1. Component map



Processes and binaries

Binary	Artifact	Role
`p4d`	Preloaded by `libp4shim.so`	Perforce server; unaware of the cache
`p4-cache`	`rust/target/release/p4-cache`	Long-running daemon: upload, restore, evict, delete, access-log, license sampling/audit
`p4-cache-verify`	`rust/target/release/p4-cache-verify`	Offline checkpoint/journal verifier; shares config with daemon
`p4-cache-manifest-summary`	Same binary tree	Operator inspection tool for the manifest state
`license-tool`	Vendor-only, gated behind the `sign` feature	Ed25519-signs license envelopes; daemon embeds the verification half only
`integrity-stamp`	Vendor-only	Stamps release artifacts so the daemon can self-verify at startup

Workspace layout

Six workspace crates under `rust/`:

- `p4cache` — daemon binary, library code, and the two operator tools
- `p4shim` — `libp4shim.so` cdylib
- `shared` — config types, protocol types, error types, `FileEntry` codec, shared utilities
- `license` — Ed25519 license envelope: claims, signing, verification, source loading
- `license-tool` — vendor CLI for signing license files (separate workspace member so the daemon never pulls signing code)
- `integrity-stamp` — vendor CLI for stamping built artifacts with a verification digest

State directories

All daemon-owned state under `$DEPOT/.p4cache/`:

- `manifest.redb` — persistent file manifest, dirty queue, by-path and by-timestamp indexes, persistent upload-retry deadlines
- `shim.sock` — Unix stream socket for FETCH/STAT/DELETE protocol
- `access.sock` — Unix datagram socket for access-log events
- `p4-cache-failed-ops.log` — JSONL ledger of terminal upload/delete failures
- `shim-failed-deletes.log` — JSONL breadcrumb for deletes that couldn't reach the daemon; drained on daemon restart
- License audit log — JSONL ledger of license-sampling decisions, used by the operator-side reconciliation tool

The reserved `.p4cache/` subtree is never served as depot content.

2. Request flow — cold read (the hot path)

```
1. p4d calls open("path/in/depot/file.uasset", O_RDONLY)
2. libp4shim's open hook fires (LD_PRELOAD); calls real open()
3. Real open returns ENOENT (file is cold)
4. Shim sends "P4CACHE/1 FETCH path/in/depot/file.uasset\n" on shim.sock
5. Daemon's shim_server task:
   a. Reads request, parses with versioned protocol header
   b. Authenticates peer via SO_PEERCRED against shim_allowed_uids
   c. Looks up the manifest entry for the relative path
   d. If state is Clean and storage_key resolves, dispatches to storage backend
6. Storage backend (Azure/S3/GCS/NFS):
   a. Issues GET (parallel range GETs for large files when chunk_threshold and
chunk_threads allow)
   b. Streams response into a temp file in the depot's cache area
   c. BLAKE3-256 hashes the bytes inline during download (no second pass)
   d. Verifies hash against manifest entry's stored hash
   e. fsync(temp_fd), rename(temp, target), fsync(parent_dir)
   f. assert_regular_file (S_ISREG) post-rename
7. Daemon replies "OK <size>\n" to the shim
8. Shim returns "do nothing" — libp4shim's open hook returns control to p4d, which retries
the real open() and gets the now-present file
9. p4d reads the file normally; subsequent reads hit the local hot tier
```

Failure modes — each enumerated in the configuration and observable:

- Hash mismatch (or `enforce_content_hash = true` and no stored hash): restore fails closed; daemon returns ERR to the shim
- Backend NotFound on primary, secondary configured: secondary attempted; success treated as primary-cache-miss
- Backend transient error: retry policy (initial + 3 retries with exponential backoff); after exhaustion, terminal failure recorded in `p4-cache-failed-ops.log`
- Daemon unreachable: shim falls through to real libc; `p4d` sees ENOENT and reports normally

---

3. Request flow — write (submit / integrate)

```
1. p4d writes a new depot file via standard libc calls
2. fanotify watcher (registered with FAN_CLASS_CONTENT | FAN_CLOSE_WRITE on the depot
subtree)
   receives CLOSE_WRITE event when p4d closes the new file
3. Watcher authenticates the writer:
   - Pulls writer PID from the fanotify event
   - SO_PEERCRED on the fanotify fd identifies the writer task's UID
   - Checks against watcher_allowed_uids
   - Falls back to fd-owner check if writer task has already exited (counter:
p4cache_watcher_auth_by_fd_owner_total)
   - Rejects events from non-allowlisted UIDs (counter:
p4cache_watcher_rejected_writer_total)
4. Watcher enqueues the file for upload via the dirty queue in the manifest
5. Engine's upload coordinator (count: upload_threads) picks up the dirty file:
   a. Computes BLAKE3-256 hash
   b. Uploads to primary backend
   c. On success, marks manifest entry Clean and stores the hash
   d. On failure, retries with backoff; after retry budget, records terminal failure
```

The write path does not block `p4d`. The file is local-fast for the entire write; the upload happens asynchronously in the daemon. `p4d` is never aware that anything happened.

---

4. Manifest

Schema

`FileEntry` per depot-relative path, stored in redb (embedded B-tree, single-writer with `fsync(2)` on commit):

```
FileEntry {
    size:         u64,
    last_access:  u64,        // unix epoch seconds
    created_at:   u64,        // unix epoch seconds
    state:        u8,         // Clean=0, Dirty=1, Uploading=2
    content_hash: Option<[u8; 32]>,   // BLAKE3-256
    storage_key:  String,     // depot-relative or backend-prefixed
}
```

Binary codec is hand-written little-endian with an explicit version byte. Three versions coexist on disk:

- **v-pre-magic** (legacy, pre-F3): no version byte; magic followed directly by fields. Decoder accepts; treated as v0.
- **v0**: magic + version=0 + fields (no hash). Used when `content_hash` is None.
- **v1**: magic + version=1 + fields + hash. Used when `content_hash` is Some.

The encoder picks the lowest faithful version: a `FileEntry` without a hash stays on disk in v0 layout, so an older daemon can still read it. Once v1 lands for a row, that row is hash-protected: rolling back to a pre-F3 daemon refuses the entry rather than silently misparsing it (the tripwire property).

Tables in redb

- `files_by_path` — primary key, full FileEntry
- `dirty_queue` — FIFO of paths needing upload; insertion-ordered
- `lru_by_timestamp` — secondary index keyed on `last_access` for eviction
- Schema-migration table for codec version coordination

Single-writer model with single-tx writes. The redb commit rate dominates manifest throughput; benchmarks show 6–8K mark_clean/mark_dirty operations per second sustained, well above production write rates (~45 file-writes/s observed in stress tests).

Cache eviction

Triggered when cache size exceeds `max_cache_bytes` (or derived from `max_cache_gb`):

- Target: bring cache below `low_watermark_bytes` (defaults to 80% of max)
- Eviction target: `max(1, low_watermark * 90%)`
- Policy: LRU by `last_access` via the secondary index
- Eviction removes the local file; the manifest entry stays (state=Clean, storage_key still resolves)
- Subsequent read of the evicted path triggers the cold-read path above

---

5. Storage backends

Trait

```
#[async_trait]
trait StorageBackend: Send + Sync {
    // Object-oriented core — backends implement these four.
    async fn upload_object(&self, key: &str, local: &Path,
                           req: &UploadRequest)   -> Result<ObjectMetadata, StorageError>;
    async fn download_object(&self, key: &str, local: &Path,
                             req: &DownloadRequest) -> Result<ObjectMetadata,
StorageError>;
    async fn head_object(&self, key: &str)        -> Result<ObjectMetadata, StorageError>;
    async fn delete_object(&self, key: &str,
                           req: &DeleteRequest)   -> Result<(), StorageError>;

    // Plain wrappers — every backend gets these for free via default impls.
    async fn upload(&self, key: &str, local: &Path)   -> Result<u64, StorageError> { … }
    async fn download(&self, key: &str, local: &Path) -> Result<u64, StorageError> { … }
    async fn download_with_metadata(&self, key: &str, local: &Path)
                                    -> Result<ObjectMetadata, StorageError> { … }
    async fn exists(&self, key: &str)                 -> Result<bool, StorageError> { … }
    async fn remove(&self, key: &str)                 -> Result<(), StorageError> { … }

    fn name(&self) -> &'static str;
    fn capabilities(&self) -> StorageCapabilities { StorageCapabilities::default() }
}
```

Single trait. Four implementations: Azure, S3, GCS, NFS. Each is feature-gated; an `nfs`-only build contains no cloud SDK code.

The object-oriented core (with `UploadRequest`/`DownloadRequest`/`DeleteRequest`/`ObjectMetadata` types) surfaces metadata, optional ETag preconditions, storage-class hints, and the inline BLAKE3 content hash (P-010) without smuggling them through ad-hoc transport code. `StorageCapabilities` advertises per-backend support for metadata, range reads, conditional reads/writes, storage classes, custom endpoints, and custom CAs.

`storage/mod.rs` also exposes shared helpers every backend funnels through: `download_to_temp_then_rename` (serial path), `download_parallel_to_temp` (parallel range-GET driver — collapses the per-backend orchestration that used to be duplicated three times), `pwrite_chunk_at`, `plan_ranges`, `hash_completed_temp_file`, `unique_temp_path`, `create_temp_for_write_no_follow`, `assert_regular_file_at`, and `fsync_parent_dir_async`.

Backend-specific notes

Azure Blob Storage

- SDK: `azure_storage_blobs` 0.21 + `azure_identity` 0.21
- Credentials: SAS, SharedKey, or Azure SDK default credential chain (managed identity, CLI login, env)
- Large restores use parallel block range GETs when `restore_chunk_threshold_mb > 0` and `restore_chunk_threads > 1` (concurrency bounded by semaphore — recently hardened to acquire-before-spawn after audit finding P-005)
- Custom endpoints supported (private Azure endpoints) with mandatory `endpoint_pin = true`

AWS S3

- SDK: `aws-sdk-s3` 1.x + `aws-smithy-http-client` with rustls

- Credentials: static or SDK default chain
- SSE-S3 (`AES256`) supported via `server_side_encryption = true`
- S3-compatible services (MinIO, Wasabi, etc.) via endpoint + `endpoint_pin = true`
- `verify_ssl = false` rejected by the runtime; use `ca_cert_path` instead

Google Cloud Storage

- SDK: `google-cloud-storage` 1.8 + `google-cloud-auth` 1.6
- Credentials: service account JSON via `credentials_file` or Application Default Credentials
- Billing-project header support via `project_id`
- `verify_ssl = false` rejected by the SDK-backed backend

NFS

- Pure POSIX filesystem operations
- `path_prefix` validated as a depot-relative safe path
- Storage keys rejected if they're absolute, contain traversal, or enter the reserved `.p4cache` directory
- No SDK dependency

Primary + secondary

Both `primary` and `secondary` use the same backend trait. Uploads always go to primary; secondary is read-fallback only, attempted on `NotFound` from primary. Useful for cloud-migration scenarios (the old NFS still has files the new bucket hasn't received yet) and for hybrid deployments. Not a mirrored write target.

---

6. Access log pipeline

```
shim ── datagram ──> .p4cache/access.sock ──> receiver ──> bounded queue ──> writer task
──> ES or Postgres
                                                                          |
                                                                          ▼
                                                                  (if sink down)
                                                                  on-disk spool
                                                                  replayed oldest-first
```

Why two distinct drop boundaries

Pre-spool drop — the bounded in-process queue between socket and writer (sized by `access_max_pending_entries`). When full, `try_send` drops the datagram immediately. The on-disk spool sits *after* this queue and does not absorb pre-queue overflow. Tracked by `p4cache_access_log_dgram_dropped_total`.

Post-spool drop — when the spool reaches `access_spool_max_bytes`, new undeliverable batches are dropped. Tracked separately.

This is explicit: every loss surface is named, configurable, observable. Operators tune `access_max_pending_entries` based on observed drop rates during burst load.

Shim-side batching

The shim batches access events in a 32 KiB thread-local buffer and flushes via `sendto(2)`:

- When the next event would overflow the buffer

- When the buffer reaches ~30 KiB high-water mark
- When the oldest unflushed event has aged beyond 1 second (low-volume threads don't sit on stale events)
- When the thread exits (Drop on the thread-local buffer drains the remainder)

Without the age and thread-exit triggers, short-lived `p4d` worker threads would silently disappear from the audit feed. With them, every event reaches the daemon's spool/retry contract within a bounded window.

---

7. High availability

p4-cache does not invent a new HA model. It fits inside Perforce's existing topology.

Single-site, single-server:

- One daemon, one shim, one hot tier (NVMe local to the server), one cold tier (object store)
- Failure of the daemon: `p4d` falls through to real libc on cold-read attempts and continues serving locally-resident files normally
- Failure of the cold tier: cold reads fail; hot reads continue normally
- Failure of `p4d`: nothing changes about p4-cache; the daemon waits for `p4d` to come back

Commit-edge replication (the standard Perforce HA pattern):

- Each `p4d` instance (commit, edge, forwarding replica) runs its own p4-cache daemon
- Each has its own NVMe hot tier
- All share the same cold tier (object store)
- Perforce journal replication handles consistency at the application layer
- Failover is the same operation as today: cut over to the replica; the replica's p4-cache serves cold files from the shared object store

What p4-cache does *not* do:

- No write-path replication. Writes are propagated through Perforce's existing journal mechanism, which is the correct layer.
- No quorum protocol. The manifest is single-writer (one daemon per `p4d` instance).
- No leader election. The deployment topology determines roles, not p4-cache.

This is the answer storage architects want to hear: "We don't reinvent your HA story. Commit-edge replication continues to work the way it does today."

---

8. Failure modes and recovery

Daemon restart

- Manifest is the source of truth on restart
- Startup scan (when not in `read_only` mode):

- Empty manifest → adopt local files as bootstrap and queue them for upload - Non-empty manifest → adopt untracked local files as Clean cache residents

- `skip_startup_scan = true` skips the disk walk (useful on very large depots after the first bootstrap)
- Shim's failed-delete breadcrumb log drained into the daemon's failed-ops log

- Dirty queue resumed from manifest

fanotify queue overflow

A burst of writes that overflows the kernel's fanotify queue triggers recovery: the daemon's `trigger_full_rescan` calls `scan_untracked_files_blocking`, which now reconciles both untracked files and previously-tracked files with stale state (the QA-pass `Q-p4cache-006` "skips already-tracked files" gap is closed). Overflow events are coalesced behind a cooldown window; one rerun is queued if another overflow fires while a rescan is in flight. A manual `p4 verify` remains the operator-side authoritative reconcile if hash mismatches surface elsewhere.

Configuration reload

The daemon polls its JSON config file's mtime every ~30 seconds. On change:

- Parse + validate the new config
- On success, apply field-by-field; restart-required fields produce warnings
- On failure, keep the previous known-good config; emit `tracing::warn!`

Hot-reloadable fields include stats intervals, cache size limits, UID allowlists, retry policies, and verbose logging. Restart-required fields include `depot_path`, backend configuration, socket modes, and access-log sink configuration. Editors that swap-and-rename (vim, `sed -i`) work because the daemon polls the path, not the inode.

Manifest corruption

redb's single-tx write model means partial writes don't happen — every commit is atomic at the storage layer. If the underlying disk corrupts the redb file, the daemon refuses to start and emits a structured error. Recovery: restore the manifest from backup, or rebuild from the depot tree and the cold tier (the manifest-rebuild path requires a verifier pass; documented in OPERATIONS.md).

---

9. Observability

Prometheus textfile metrics

Written to `metrics_file` (when configured), refreshed every `metrics_interval_secs` (default 15s):

- `p4cache_uploads_total{result="success|failure"}`
- `p4cache_restores_total{result="success|failure"}`
- `p4cache_deletes_total{result="success|failure"}`
- `p4cache_evictions_total`
- `p4cache_manifest_counter_drift_total`
- `p4cache_files{state="dirty|uploading|clean|total"}`
- `p4cache_cache_bytes` / `p4cache_cache_max_bytes`
- `p4cache_shim_rejected_peer_total` / `p4cache_shim_legacy_unprefixed_total`
- `p4cache_watcher_rejected_writer_total` / `p4cache_watcher_auth_by_fd_owner_total`
- `p4cache_access_log_dgram_dropped_total` / `p4cache_access_log_peer_rejected_total`
- `p4cache_license_managed_bytes` — sampled managed-capacity attestation surface

- `p4cache_license_soft_ceiling_bytes` — warn-at threshold from the embedded license claims
- `p4cache_license_hard_ceiling_bytes` — refuse-uploads threshold from the embedded license claims

Failure counters record terminal failures only — uploads increment after retry exhaustion, not on transient errors that keep the file queued.

Structured tracing

`tracing-subscriber` with env-filter. Output to stdout, redirectable to `log_file`. Daily age rotation; optional size-based rotation via `log_rotate_max_mb`. Configurable retention.

Failed uploads have a separate log (`p4-cache-failed-uploads.log`) with its own rotation policy.

Periodic stats

`stats_interval_secs` controls a periodic summary log line: counts of files in each state, cache size, recent throughput. 0 disables.

---

10. Performance characteristics

Numbers from `docs/perf/baseline.md`, captured via criterion benchmarks:

BLAKE3 hashing (single-threaded, Intel Xeon Gold 6444Y):

- 1 MiB: 5.43 GiB/s
- 100 MiB: 3.69 GiB/s
- 1 GiB: throughput sustained at ~3.7 GiB/s with file-cache warmth

Manifest writes (single-tx redb commits, fsync per commit):

- mark_clean: 7K elem/s at 1K entries, 7K at 100K entries (essentially flat — manifest doesn't get slower with size)
- mark_dirty: 7K/s at 1K, 6K/s at 100K
- pop_lru_and_remove: 8.5K/s at 1K, 6.5K/s at 100K

Manifest throughput is bounded by redb's single-writer commit rate. Observed production peak write rate (45 files/s) is well under the 6K elem/s manifest ceiling — the manifest is not the bottleneck.

Restore latency depends on backend, file size, and chunk parallelism. Reproducible benchmark and stress scripts in `rust/tests/`:

- `p4_storage_benchmark.sh` — upload/download throughput across backends
- `p4_cache_read_stress.sh` — Azure read-heavy stress harness
- `p4_azure_nfs_fallback_test.sh` — primary→secondary fallback verification
- `p4_commit_peak_hour_stress.sh` — write-burst simulation (CLOSE_WRITE pressure on the fanotify queue)
- `p4_read_stress_nocache.sh` — cold-only read pattern with the hot tier disabled
- `p4_submit_storage_compare.sh` — apples-to-apples submit-path comparison vs. an unaccelerated baseline
- `compare_benchmarks.sh` — criterion baseline regression check

All scripts support Azure-backed runs and default to a local test `p4d` port instead of `1666`. The prospect's environment should produce the numbers that matter for *their* workload; the methodology is documented so they can reproduce.

11. Build and deployment

Source layout:

- `rust/p4cache` — daemon binary (`p4-cache`), library, two operator tools (`p4-cache-verify`, `p4-cache-manifest-summary`)
- `rust/p4shim` — `libp4shim.so` cdylib
- `rust/shared` — config types, protocol types, error types, `FileEntry` codec, shared utilities
- `rust/license` — Ed25519 license envelope, claims, signing (vendor-only), verification (daemon)
- `rust/license-tool` — vendor CLI that signs license files (daemon never links signing code)
- `rust/integrity-stamp` — vendor CLI that stamps shipped artifacts for self-verification at daemon startup
- `rust/tests` — bash scripts for benchmarks and stress harnesses
- `rust/p4cache/fuzz` — sibling crate with `cargo fuzz` targets (excluded from the workspace so `cargo build` doesn't pull in `libfuzzer-sys`)

Toolchain: rustc 1.88+ (workspace MSRV pinned), Rust edition 2024. Stable channel.

Build:

```
cd rust && cargo build --release
```

Per-cloud builds reduce binary size and attack surface:

```
cargo build --release --no-default-features --features azure  # ~10 MiB
cargo build --release --no-default-features --features s3
cargo build --release --no-default-features --features gcs
cargo build --release --no-default-features --features nfs    # ~9 MiB
```

CI publishes per-cloud release artifacts under `p4-cache-<feature>-<git-sha>` naming.

Debian package:

```
./packaging/build-deb.sh
# produces packaging/out/p4-cache_<version>_<arch>.deb
```

Package contents include `/usr/bin/p4-cache`, `/usr/lib/p4-cache/libp4shim.so`, `/lib/systemd/system/p4-cache.service`, and an example config at `/etc/p4-cache/config.json.example`. The systemd unit runs as `perforce`, expects `daemon = false` in config (so systemd owns supervision), and grants `CAP_SYS_ADMIN` via `setcap`.

12. Licensing subsystem

The runtime carries an embedded license envelope that gates managed-capacity claims, ceiling enforcement, and forensic watermarking. The subsystem is architecturally relevant — not just a commercial wrapper — because every cold-tier write carries identity metadata that becomes part of audit evidence in incident response.

Envelope shape

Ed25519-signed JSON envelope (RFC 8785 JCS canonicalization so signer and verifier byte-agree regardless of map ordering). Claims include `customer`, `instance`, `issued_at`, `expires_at`, `maintenance_expires_at`, `managed_capacity_soft_ceiling`, `managed_capacity_hard_ceiling`, and a `customer_secret` used to derive the deployment watermark. Verification keys are embedded into the daemon at build time via a `build.rs`-generated `vendor_keys.rs`.

Components

Crate / module	Role
`rust/license`	Envelope, claims, signing (vendor-only via `sign` feature), verification, source loading. Daemon links verify-only.
`rust/license-tool`	Vendor CLI: generate keypairs, sign envelopes. Never shipped to customers.
`rust/integrity-stamp`	Vendor CLI: stamps release artifacts so the daemon can self-verify at startup.
`p4cache/src/license/load.rs`	Reads + verifies the on-disk envelope at daemon startup.
`p4cache/src/license/policy.rs`	Enforces ceilings, expiry, maintenance windows; emits the three `p4cache_license_*` metrics.
`p4cache/src/license/watermark.rs`	Derives a forensic watermark from `customer_secret` + `instance` for every cold-tier write.
`p4cache/src/license/sampler.rs`	Periodic managed-capacity sampling for the `managed_bytes` metric.
`p4cache/src/license/audit_log.rs`	JSONL ledger of sampling/policy decisions for operator-side reconciliation.
`p4cache/src/license/banner.rs`	Startup banner identifying customer/license/expiry.
`p4cache/src/license/integrity.rs`	Cross-check of the installed binary against the `integrity-stamp` digest.

Forensic watermark — where it shows up

Every cloud-backend upload tags object metadata with the watermark (`x-amz-meta-*` on S3, blob metadata on Azure, custom metadata on GCS). The SDK `User-Agent` (where the SDK exposes it — currently S3 and Azure) carries an `application_id`/`AppName` derived from the same secret. The anonymous `Watermark::empty()` short-circuits both, used by the integration-test corpus so test artifacts don't carry watermarks.

Practical consequences for an architecture review:

- The watermark is **passive** — it does not affect read/write semantics, throughput, or correctness.
- Anyone with backend-read access can extract the watermark from any object's metadata. This is the design intent: a leaked or exfiltrated cold-tier blob is traceable to the licensed deployment that wrote it.
- Disabling the watermark requires either a custom build or a license envelope with the empty-watermark variant. Neither is shipped to customers.

Failure modes

- **Missing or invalid license:** daemon refuses to start, logs a structured error pointing at the license-load path.
- **Expired license:** daemon continues serving but emits an `expired` banner on every startup and via tracing periodically. Uploads continue; the `expires_at` field is informational unless the operator has also configured a hard refusal policy.
- **Soft-ceiling breach:** warning log + `p4cache_license_managed_bytes` exceeds `p4cache_license_soft_ceiling_bytes` in Prometheus; no operational effect.
- **Hard-ceiling breach:** uploads refuse; reads continue. The metric exposes the breach so operator alerting can fire.
- **Stamp/binary mismatch:** `integrity::verify` fails at startup; daemon refuses to start.

---

13. Questions an architecture review will ask

Pre-answers to the questions every review touches:

"How does this affect `p4d`?" It doesn't, beyond the LD_PRELOAD. No patches, no protocol changes, no client software. The shim guarantees panic safety via `catch_unwind` (ADR-0001); a shim bug falls through to real libc rather than crashing `p4d`.

"What happens when the daemon dies?" Cold reads fail; hot reads continue normally. `p4d` survives. On daemon restart, manifest is the source of truth; the daemon adopts local state and resumes the dirty queue.

"What's the failure mode if the cold tier becomes unavailable?" Cold reads fail; hot reads continue. Writes are queued in the dirty queue until the cold tier recovers; retry budget exhaustion records terminal failures in `p4-cache-failed-uploads.log` for operator review.

"Can I roll this out incrementally?" Yes. `read_only = true` deploys the shim and daemon without any upload or fanotify activity — restores still work for cold files that the operator pre-staged with a separate one-time tool. Use for migration windows and eval deployments.

"Can I roll this back?" Yes. Stop the daemon, remove the `LD_PRELOAD`, restart `p4d`. The depot tree is whatever's locally resident; the cold tier remains intact in case you want to come back. A bulk-rehydrate tool runs against the manifest and pulls all entries into the hot tier if you need a clean exit.

"What if there's a bug in your code that causes data loss?" Three defenses:

1. The cold tier is the durable copy. Hot-tier corruption is recoverable by re-downloading from the cold tier.
2. Content-hash verification on every restore catches bit-flips in the cold tier itself.
3. Perforce's own journal replication provides a parallel durability path at the application layer.

Worst plausible case: a daemon bug corrupts the manifest. The manifest can be rebuilt from the depot tree and the cold tier via the verifier tool. No depot data is lost.

"What about the LD_PRELOAD risk?" LD_PRELOAD shims are real risk. Three defenses:

4. Every libc hook is wrapped in `ffi_guard` (`std::panic::catch_unwind`); a Rust panic in shim code falls through to real libc rather than crashing `p4d`.
5. The workspace builds with `panic = "unwind"` so `catch_unwind` actually fires (ADR-0001).
6. 192 unsafe blocks in the shim, every one carrying a `// SAFETY:` comment justifying the invariant. Fuzz harness on the protocol parser. Lint-time prevention of unsafe-op without explicit block via `unsafe_op_in_unsafe_fn = "deny"`.

"How do you handle the audit trail if Postgres is unreachable?" Exponential backoff with retry budget. Optional on-disk spool (`access_spool_dir` + `access_spool_max_bytes`) absorbs batches during outages and replays oldest-first when the sink recovers. Auto-disable after configurable consecutive failures with warning rate limit. Two distinct drop boundaries (pre-spool queue overflow vs. post-spool spool overflow), both observable in Prometheus.

"Can we run this in an air-gapped environment?" Yes. No telemetry, no phone-home. The daemon contacts only the customer-configured backends. NFS backend has no external dependency at all. The Postgres audit sink works on-prem.

Document control

- Version: 1.0
- Maintained against the source repository
- Use during 60–90 minute architecture review sessions; one page-per-section is the intended cadence
- Pair with the Security & Compliance Brief and the Operations Guide for a full picture