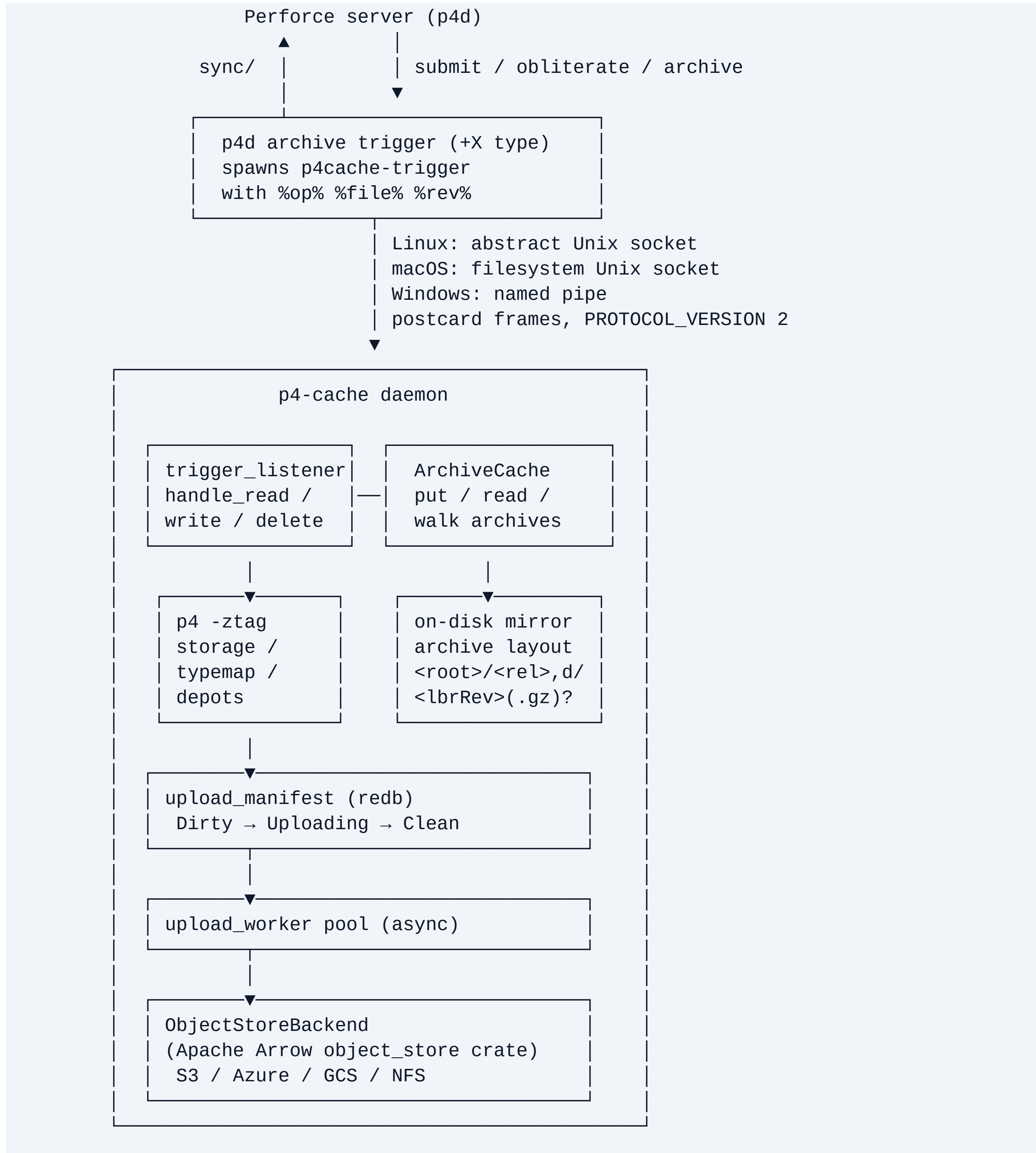
p4-cache — Architecture Deep-Dive

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



```
                     |
            in-process AccessEvent channel
                     |
                     ▼
         ┌──────────────────────────┐
         │ access_log → ES / Postgres │
         │     + local spool          │
         └──────────────────────────┘
```

Processes and binaries

Binary	Artifact	Role
`p4d`	Stock Perforce server; archive trigger installed via `p4 archive`	Drives `p4cache-trigger` on every `+X` archive op
`p4cache-trigger`	`rust/target/release/p4cache-trigger`	Per-call static binary; postcard over local IPC; no async runtime; ~1 ms warm
`p4-cache`	`rust/target/release/p4-cache`	Long-running daemon: serves the trigger socket, manifest + async upload, eviction, delete, access-log, license sampling/audit. Also hosts the operator `cache rescan` / `manifest summary` subcommands.
`p4-cache-verify`	`rust/target/release/p4-cache-verify`	Standalone offline checkpoint/journal verifier; shares config with the daemon
`p4-cache-manifest-summary`	`rust/target/release/p4-cache-manifest-summary`	Standalone operator inspection / drift-quarantine tool
`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 operator subcommands
- `p4cache-trigger` — per-call archive trigger binary (static, no async runtime)
- `shared` — config types, wire-protocol types, error types, trigger transport, 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

Daemon-owned state lives under the configured `state_dir`:

- `v2/upload_manifest.redb` — durable async-upload state machine (Dirty/Uploading/Clean) plus the delete queue; the eviction and saturation gates read it
- `archive_catalog.redb` — local cache of `(depot_path, lbr_rev) → {md5, gzipped}` that keeps `p4 -ztag storage` off the read hot path; `db.storage` stays authoritative, so this entry is rebuildable and never pruned on eviction
- Trigger socket — Linux abstract address `\0p4cache.trigger.v1`; macOS filesystem socket; Windows named pipe `\\.\pipe\p4cache.trigger.v1`
- License audit log + `license-audit.anchor.json` — tamper-evident NDJSON chain anchored across restarts
- Operator log, optional rotation siblings

The cached archives themselves live at the mirrored p4d archive path (`<archive_root>/<rel>,d/<lbrRev>(.gz)?`), not under `state_dir`. The reserved `.p4cache/` subtree is never served as depot content.

---

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

```
1. A `p4 sync` or `p4 print` references a `+X` archive that isn't
   resident on the p4d host
2. p4d invokes the archive trigger:
     p4cache-trigger read <depot_file> <%rev%>
3. The trigger connects to the daemon's trigger socket and exchanges
   a `Hello` (PROTOCOL_VERSION 2, advertises SCM_RIGHTS fd-passing
   on Unix), then sends a postcard `TriggerRequest { op: Read, ... }`
4. The daemon's trigger_listener:
   a. Authenticates peer via SO_PEERCRED against the configured
      `trigger_allowed_uids`
   b. Validates depot_path + revision (refuses traversal, NUL, LF/CR,
      `-`-prefix argv tricks)
   c. Calls ArchiveCache::locate_archive at the mirrored path
      `<archive_root>/<rel>,d/<lbrRev>(.gz)?`
   d. On miss, dispatches to the cold-restore coalescer + storage backend
5. ObjectStoreBackend (S3 / Azure / GCS / NFS, all via the Apache Arrow
   object_store crate):
   a. Streams the object to a temp file in the cache dir
   b. Inline-decompresses (if gzipped) and streaming-MD5-hashes the
      bytes as they're written — no second pass, no in-memory Vec
   c. Verifies MD5 against `db.storage.digest` (queried via
      `p4 -ztag storage`)
   d. fsync(temp_fd), rename(temp, target), fsync(parent_dir)
6. The daemon either streams the bytes back over the socket or, when
   both ends advertise it, hands the trigger a file descriptor via
   `sendmsg(2)` + SCM_RIGHTS (zero-copy)
7. The trigger writes the archive bytes to p4d's stdout and exits 0
8. p4d serves the file to the client; subsequent reads on this host
   skip the cold-restore path because the archive is now resident at
   the mirrored path
```

Failure modes — each enumerated and observable:

- MD5 mismatch against `db.storage.digest`: restore fails closed; daemon returns Transient; trigger exits 3
- Backend NotFound: daemon returns NotFound; trigger exits 2; p4d surfaces ENOENT to the client
- Backend transient error: structured `Transient { code: BackendUnreachable | BackendThrottled | LocalResource | UploadPending, reason }`; trigger exits 3
- Daemon unreachable: trigger exits 1 immediately; p4d surfaces the failure to the client (`systemd Restart=always` is mandatory in production)
- Cache corruption: `p4-cache cache rescan` cross-references `db.storage` and quarantines or deletes drift

---

3. Request flow — write (submit / integrate)

```
1. p4d submits a new rev of a `+X` archive and invokes:
     p4cache-trigger write <depot_file> <%rev%>
   piping the archive bytes to the trigger's stdin
2. The trigger streams the bytes over the trigger socket in a
   length-prefixed postcard frame (Write op + payload bytes)
3. The daemon's handle_write:
   a. Resolves lbr_type from the typemap snapshot (the new rev isn't
      yet in `db.storage`); gzip on the write path unless the file is `+F`
   b. Atomically writes to the mirrored archive path via temp + fsync +
      rename + fsync(parent); records `user.p4cache.size` xattr (the
      decompressed size, used so reads of >4 GiB files report a correct
      payload_bytes without re-decompressing)
   c. Marks the manifest entry Dirty in redb (Dirty → Uploading → Clean)
   d. Returns Ok to the trigger; the trigger exits 0; p4d sees the
      submit complete with no upload latency on the critical path
4. The upload_worker pool drains the Dirty queue asynchronously:
   a. Claims a row (Dirty → Uploading)
   b. Streams the local archive to the object_store backend
   c. On success, transitions to Clean
   d. On failure, reverts to Dirty with backoff for retry
5. On daemon restart, any rows left in Uploading are reverted to Dirty
   so a fresh worker re-tries; the upload is idempotent against the
   bucket key
```

The write path does not block `p4d`: the trigger returns as soon as the bytes are durably in the local cache. The upload happens asynchronously in the daemon. `p4d` is never aware that a cold tier exists.

---

4. Upload manifest and cache state

What state lives where

The design pushes the `(depot, rev) → archive` index back into the filesystem path itself and uses `p4d`'s `db.storage` (already replicated by SDP) as the source of truth for refcount, digest, and lbrType. The daemon does not maintain a parallel content-addressed manifest. Two narrow redb stores remain, and neither is that index:

- **`upload_manifest.redb`** — the durable async-upload state machine for files the daemon has accepted on the write path but not yet pushed to the bucket (Dirty → Uploading → Clean), plus crash recovery, the eviction gate, and the saturation gate.
- **`archive_catalog.redb`** — a local cache of `(depot_path, lbr_rev) → {md5, gzipped}` populated on writes and restores. It keeps `p4 -ztag storage` off the read hot path. `db.storage` stays the replicated, authoritative digest source; the catalog is never pruned on eviction, and `p4 -ztag storage` is the cold / cross-daemon fallback when an entry is absent.

Upload manifest schema

One redb table, one row per cache file the daemon has written:

```
UploadRow {
    rel_path:     String,     // depot-relative archive key
    lbr_rev:      String,     // <change>.<rev>[.shelf]
    state:        u8,         // Dirty=0, Uploading=1, Clean=2
    enqueued_at:  u64,
    attempts:     u32,
    last_error:   Option<String>,
}
```

A sibling `delete_queue_v1` table tracks bucket-keys that need removing (the daemon resolves refcount against `db.storage` and only deletes the blob when the count reaches zero).

Crash recovery

On startup, any row in `Uploading` reverts to `Dirty` so a fresh worker re-tries; uploads are idempotent against the bucket key.

Cache eviction

The eviction loop walks the cache archive tree directly via `ArchiveCache::walk_archives`:

- Tick cadence: 5 s, plus an async signal-driven wake from the write path when the watermark is crossed
- Policy: age-based, oldest mtime first (the `+X` trigger doesn't see hot reads, so an LRU access timestamp would be unreliable)
- Target: bring total bytes below the eviction target (default 90% of the low watermark)
- Respects `Dirty` / `Uploading` rows — eviction never touches bytes that haven't reached the bucket yet
- Saturation back-pressure: `handle_write` refuses new writes (with `LocalResource` Transient) when the cache is full of Dirty bytes and eviction can't make progress

---

5. Storage backends

Trait

```
#[async_trait]
trait StorageBackend: Send + Sync {
    // Object-oriented core.
    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 — 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. One production implementation: `ObjectStoreBackend`, a thin wrapper over the Apache Arrow `object_store` crate. Backend choice (S3 / Azure / GCS / NFS) is a runtime config knob; the same code path serves every cloud.

The object-oriented core (with `UploadRequest`/`DownloadRequest`/`DeleteRequest`/`ObjectMetadata` types) surfaces metadata, optional ETag preconditions, storage-class hints, and inline content-hash computation 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 the backend funnels through: `download_to_temp_then_rename` (serial path), `download_parallel_to_temp` (parallel range-GET driver), `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

AWS S3 (and S3-compatible)

- Driven through `object_store`'s `aws` feature
- Credentials: static (`access_key` / `secret_key`) or the AWS default credential chain
- S3-compatible services (MinIO, Wasabi, etc.) via `endpoint` + `endpoint_pin = true`
- `verify_ssl = false` is rejected at config-load time

Azure Blob Storage

- Driven through `object_store`'s `azure` feature
- Credentials: SAS, SharedKey (`account_name` / `account_key`), or the Azure default credential chain
- Custom endpoints (private Azure endpoints) require `endpoint_pin = true`

Google Cloud Storage

- Driven through `object_store`'s `gcp` feature
- Credentials: service account JSON via `credentials_file` or Application Default Credentials
- `verify_ssl = false` is rejected at config-load time

NFS (LocalFileSystem)

- Pure POSIX filesystem operations via `object_store`'s `LocalFileSystem`
- `path_prefix` validated as a depot-relative safe path
- Storage keys rejected if they're absolute, contain traversal, or enter the reserved `.p4cache` directory

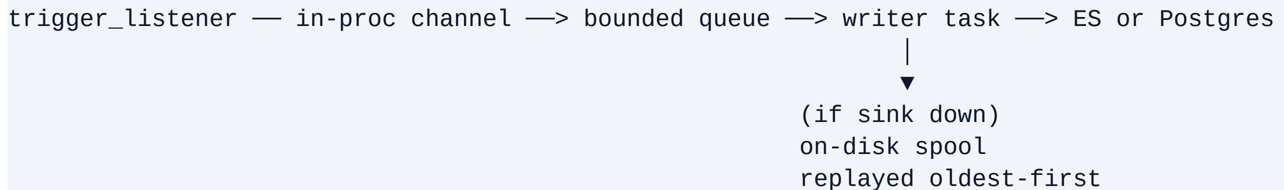
- No SDK dependency in the binary

Cloud-side layout

The cloud bucket mirrors the on-disk archive path one-for-one:
`<bucket>/<depot_path>,d/<lbrRev>(.gz)?` (no leading `//`). One object per `(depot, lbrRev)`.
Idempotent uploads on write; refcount-aware removal on delete (the daemon queries
`db.storage.lbrRefCount` and only drops the blob when the row is gone or the count is zero).

---

6. Access log pipeline



```
trigger_listener ── in-proc channel ──> bounded queue ──> writer task ──> ES or Postgres
                                                                │
                                                                ▼
                                                        (if sink down)
                                                        on-disk spool
                                                        replayed oldest-first
```

The trigger listener knows the size and path of every served archive directly, so access events flow into the writer task through an in-process channel — no datagram framing on the hot path. The trigger listener `try_send`s an `AccessEvent` per served archive op straight into the bounded queue; the ES / Postgres sinks and the on-disk spool are unchanged.

Two distinct drop boundaries

Pre-spool drop — the bounded in-process queue between the trigger listener and the writer (sized by `access_max_pending_entries`). The listener uses a non-blocking `try_send`, so when the queue is full the event is shed immediately and the served archive op is never delayed. The on-disk spool sits *after* this queue and does not absorb pre-queue overflow.

Post-spool drop — when the spool reaches `access_spool_max_bytes`, new undeliverable batches are dropped. Schema-invalid entries discarded on spool replay are tracked separately by `p4cache_access_spool_invalid_dropped_total`.

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

---

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 trigger binary, one hot tier (NVMe local to the server), one cold tier (object store)
- Failure of the daemon: archive triggers exit non-zero; p4d surfaces the failure to clients. `systemd Restart=always` is mandatory in production.
- Failure of the cold tier: cold reads fail; hot reads continue normally
- Failure of `p4d`: nothing changes about p4-cache; the daemon idles until `p4d` comes 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
- `db.storage` (the source of truth for refcount/digest/lbrType) is replicated by SDP for free — the daemon does not maintain a parallel index that could drift across hosts
- 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. Each daemon is the single writer for its own host's cache.
- 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

- The upload manifest reverts any `Uploading` rows to `Dirty` so a fresh worker re-tries; uploads are idempotent against the bucket key
- The on-disk archive tree IS the `(depot, lbrRev) → file` index — no separate manifest scan needed to rebuild it
- `p4-cache cache rescan` walks the cache, cross-references `db.storage`, and quarantines or deletes drift

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 backend configuration, socket modes, and access-log sink configuration. The typemap snapshot is also hot-reloadable via SIGHUP or a 60 s poll. 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. Because the upload manifest only tracks in-flight upload state, recovery is straightforward: the on-disk archive tree itself is the index, so the manifest can be rebuilt by re-enqueueing every local archive as Dirty.

---

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_files{state="dirty|uploading|clean|total"}`
- `p4cache_cache_bytes` / `p4cache_cache_max_bytes`
- `p4cache_up` / `p4cache_startup_seconds`
- `p4cache_trigger_protocol_version`
- `p4cache_access_spool_invalid_dropped_total`
- `p4cache_size_sidecar_write_failures_total`
- `p4cache_license_info{customer_id, customer_name, license_id, tier, source, vendor_key_id}`
- `p4cache_license_capacity_state` (0=Green, 1=Warn, 2=Grace, 3=Refused)
- `p4cache_license_expiry_state` (0=Healthy, 1=NearingExpiry, 2=Expired, 3=BeyondGrace)
- `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
- `p4cache_license_expires_at_unix`

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

Reproducible benchmark and stress scripts live in `rust/tests/`:

- `p4_v2_read_stress.sh` — read-heavy stress harness (hermetic p4d + typemap +X + trigger; baseline in `docs/perf/baseline.md`)
- `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` — backend fallback verification
- `p4_commit_peak_hour_stress.sh` — write-burst simulation
- `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
- `p4_gzip_parity.sh` — verifies the daemon's on-disk archive layout matches p4d's exactly
- `compare_benchmarks.sh` — criterion baseline regression check

Per-byte hashing throughput reference (captured 2026-04-27 on Intel Xeon Gold 6444Y; reference baseline):

- 1 MiB: 5.43 GiB/s
- 100 MiB: 3.69 GiB/s

The install-gating restore digest is streaming MD5, matched against `db.storage.digest` (p4d's own stored hash) rather than a self-referential cache-file hash that could drift. BLAKE3 backs the object-store download content-hash (`compute_blake3_file`), the license-audit MAC chain, and the integrity stamp. The current performance record is `docs/perf/baseline.md`.

Restore latency depends on backend, file size, and chunk parallelism. All scripts 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`) plus the standalone operator binaries (`p4-cache-verify`, `p4-cache-manifest-summary`), library, and the in-daemon `cache rescan` / `manifest summary` subcommands
- `rust/p4cache-trigger` — per-call archive trigger binary (`p4cache-trigger`), static, no async runtime
- `rust/shared` — config types, wire-protocol types, error types, trigger transport, 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
```

Release artifacts: the two shipping binaries `rust/target/release/p4-cache` (daemon) and `rust/target/release/p4cache-trigger` (per-call trigger), plus the standalone operator tools `p4-cache-verify` and `p4-cache-manifest-summary`.

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`, a `capacity_tib` claim (driving the soft / grace / hard ceiling state machine), and a `customer_secret` used to derive the deployment watermark and key the license-audit-log HMAC chain. Verification keys are embedded into the daemon at build time via a `build.rs`-generated `vendor_keys.rs`. Each key carries an explicit `signature_alg` field; the verifier requires both the envelope's declared algorithm and the matched key's algorithm to be `Ed25519` (defense in depth against a future downgrade). `key_id` comparison is constant-time via `subtle::ConstantTimeEq`. Past-`expires_at` envelopes and capacity values ≥ 1 EiB are rejected at parse time.

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. Discovery order: `--license <path>` flag → `P4CACHE_LICENSE` env (file must be root-owned 0600 and non-symlink, every honored override is audit-logged) → `<config_dir>/license.lic` → `/etc/p4cache/license.lic` → in-binary developer license.
`p4cache/src/license/policy.rs`	Hot-path admission guard. Four-state capacity machine (Green < 80%, Warn 80–100%, Grace 100–120%, Refused ≥ 120% of `capacity_tib × 1 TiB`); state packed into a single `AtomicU16` so the hot path never blocks. 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`	Tamper-evident NDJSON ledger. Monotonic `seq`, BLAKE3 keyed-MAC chain keyed by `customer_secret`, RFC 8785 canonical HMAC input via `serde_jcs`, cross-restart

	anchor file (`license-audit.anchor.json`) recording `(latest_seq, latest_hmac)` for truncation detection. Developer-build licenses write plaintext "unchained" lines.
`p4cache/src/license/clock_anchor.rs`	Cross-restart wall-clock anchor at `<state_dir>/license/monotonic-anchor`. Refuses startup if the clock has moved backward by more than `CLOCK_SKEW_TOLERANCE_SECS` (1 hour) — closes the "rewind the clock to load a license past its grace window" path that virtualised-time environments otherwise admit.
`p4cache/src/license/banner.rs`	Startup banner identifying customer/license/expiry.
`p4cache/src/license/integrity.rs`	Self-hash via `dl_iterate_phdr` over every `PT_LOAD` segment with `PF_X` set (covers LTO/PGO/split-text patches; matches the bytes actually executing, not what's on disk). Hard-fail at startup on mismatch; opt-out via `P4CACHE_INTEGRITY_SOFT_FAIL=1` (build-pipeline-only; the override is itself audit-logged loudly at startup).

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. Past-`expires_at` envelopes and capacity ≥ 1 EiB are rejected at parse time before policy enforcement runs.
- **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.
- **Capacity Warn (≥ 80% soft):** warning log; `p4cache_license_managed_bytes` exceeds `p4cache_license_soft_ceiling_bytes`. No operational effect.
- **Capacity Grace (≥ 100% soft):** SEV-1 log. Still serves everything.
- **Capacity Refused (≥ 120% soft):** refuses new manifest admissions and new cold-tier uploads; reads, restores of already-tracked files, and uploads of already-Dirty files keep working. Operator alerting should be wired

to the `managed_bytes / soft_ceiling_bytes` ratio so the operator sees Warn/Grace long before Refused.

- **Stamp/binary mismatch:** `integrity::verify` walks the loaded `PT_LOAD/PF_X` segments via `dl_iterate_phdr` and compares to the value baked in by `integrity-stamp`. Mismatch hard-fails the daemon at startup. The soft-fail override (`P4CACHE_INTEGRITY_SOFT_FAIL=1`) is loudly logged and audit-logged when honored.
- **Wall-clock rolled back across restarts:** the cross-restart anchor at `<state_dir>/license/monotonic-anchor` refuses startup if the current clock has moved backward by more than 1 hour relative to the last successful run. Defeats the trivial expired-license bypass available in containers and unprivileged user namespaces.
- **License-audit-log truncated:** the chain's cross-restart anchor file (`license-audit.anchor.json`) records the highest-seq HMAC after every write. A truncated tail produces a max-seq lower than the anchor; the loader logs loudly on mismatch.
- **`P4CACHE_LICENSE` env override:** loader refuses the override if the target file isn't root-owned, mode 0600, or is a symlink. Every honored override emits an audit-grade tracing event with the resolved path.

---

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 `+X` typemap modifier and the registered archive trigger. No patches, no protocol changes, no client software. The trigger is a per-call static binary; a trigger crash exits non-zero and surfaces as a stock p4d archive-op failure rather than affecting p4d's process state.

"What happens when the daemon dies?" Trigger calls exit non-zero (code 1, "daemon unreachable"); p4d surfaces the failure to the client. `systemd Restart=always` is mandatory in production. On daemon restart, any in-flight `Uploading` rows revert to `Dirty` and the worker pool resumes.

"What's the failure mode if the cold tier becomes unavailable?" Cold reads of files not resident locally fail with a Transient error. Writes succeed (they land in the local cache and the manifest marks them Dirty); the upload worker retries with backoff. Eviction respects Dirty rows, so cached bytes stay durable on the local NVMe until the backend recovers.

"Can I roll this out incrementally?" Yes. `+X` is a per-file typemap modifier — operators choose which depot paths route through the cache. Start with one project subtree, validate, then widen the typemap glob.

"Can I roll this back?" Yes. Remove the `+X` modifier from the typemap and `p4 archive` the affected files back to ordinary archive storage. The depot tree is whatever's locally resident; the cold tier remains intact in case you want to come back.

"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. Every restore is streaming-MD5 verified against `db.storage.digest` — the same hash p4d already stores. Bit-flips in the cold tier fail closed rather than reach p4d.
3. Perforce's own journal replication provides a parallel durability path at the application layer.

Worst plausible case: a daemon bug corrupts the upload manifest. The on-disk archive tree is the authoritative `(depot, lbrRev) → file` index, so the manifest can be rebuilt by re-enqueueing every local archive as Dirty.

"How does the cache get in front of p4d's archive I/O?" Through p4d's own `+X` archive trigger — a per-call static binary that p4d spawns with `<op> <file> <rev>`. There are no in-process libc hooks inside p4d. The

trigger model removes every concern about in-process hooks, panic-safety boundaries, and daemon-loaded shared objects inside the server, while letting p4d remain a stock build.

"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