TRIGLAV OS · QUINZEE · CONTRIBUTOR'S GUIDE

Volume 3: Quinzee Internals

TOMAZ STIH

FIRST EDITION



Development edition · 16 November 2026. Commands using repository paths, Make targets or sample projects require access to the Quinzee source checkout. No public installer is available. Examples use disposable local environments.

Your path through Quinzee

Follow the implementation from C++ ownership to Unix processes, sockets, durable files and external runtimes. Finish by inspecting a disposable native instance.

1. Architecture and trust boundaries
2. Source layout and conventions
3. HTTP library and asynchronous execution
4. Connection and streaming lifecycle
5. TLS, identity and DNS
6. Build System and Dependency Cache
7. Containers and runtime ownership
8. Testing and verification
9. Module host and service state
10. QHQ and libsimple
11. Deployments and native engines
12. Telemetry and Azure monitoring
13. The compatibility application
14. Service Bus and Functions execution
15. From a shell command to a running service
16. Follow a packet through names, sockets and namespaces
17. Files, transactions and what a crash can leave behind
18. One directory behind HTTPS and LDAP
19. From desired state to a real container
20. Inspect a disposable instance from Linux

Commands marked **host** run from the repository root. Each exercise identifies its prerequisites, the place to run commands and the expected result. Use a disposable environment while learning.

1. Architecture and trust boundaries

Quinzee separates HTTP transport, service semantics, privileged native-engine operation and application clients. The components communicate through explicit APIs or validated mailboxes, rather than sharing service implementation objects.

```
QHQ / SDK client -- HTTPS --> listener -- worker queue --> module host
                                                          |-- directory / ARM
                                                          |-- in-process services
                                                          |-- private durable state
                                                          |-- engine/hosting mailboxes
AMQP client -- TLS --> Service Bus worker ------------------ SQLite broker
LDAP client -- LDAPS --> slapd -- Unix socket --> shared directory
                                                 |-- dedicated PostgreSQL

operator-run helpers -- Docker --> SQL/cache and Functions/web containers
cloud gateway -- verified upstream HTTPS --> hosted application
native SQL/cache client -- verified TLS --> engine container
DNS module -- zone files --> CoreDNS --> client name resolution
telemetry exporter -- HTTPS OTLP --> optional collector/tools
```

`include/http/` exposes project-owned transport types. `include/quinzee/` exposes module hosting, authentication/resource capabilities, persistence, engine-runtime and telemetry contracts. Azure-specific behavior belongs in the modules under `src/quinzee_cloud/modules/`, not the transport library. QHQ links client code and Simple, not emulator service implementations.

Each HTTP listener owns an I/O thread. The cloud uses a bounded worker executor for service work and posts completions back to the connection's executor. The Service Bus AMQP worker serializes protocol events; a store mutex coordinates it with HTTP callers. Functions gateway exchanges use verified upstream HTTPS with bounded bodies and deadlines. Deployment processing and the native engine supervisor are additional, explicit background-work boundaries. Provider state has its own synchronization; there is no single global JSON database or universal request lock.

The cloud container never receives a Docker socket. The separately operated supervisor translates validated desired state into a fixed set of pinned engine operations. Public CA certificates go to clients; signing keys remain with their owner. Platform diagnostics exclude sensitive request content, while application telemetry submitted to a workspace remains application-controlled data.

See actual architecture (source checkout required), module contracts (source checkout required), provisioning (source checkout required) and observability (source checkout required).

Related reading

- The Book of Quinzee: Meet Quinzee and its services
- The Book of Quinzee: Run your first application
- Quinzee Administrator’s Guide: Start with a small, understood installation
- Quinzee Administrator’s Guide: Connect QHQ and make changes deliberately
- Scenario overview: Developing for Azure Without Using Azure
- Scenario overview: Learning How the Cloud Works
- Scenario overview: Building a Complete Cloud Application from Scratch

2. Source layout and conventions

Path	Responsibility
`include/http/`, `lib/http/src/`	Public HTTP API and private Asio/Beast/OpenSSL transport
`include/quinzee/`, `lib/quinzee/src/`	Host/capability contracts, helpers, persistence, engine mailbox and telemetry
`src/quinzee_cloud/`	Options, catalogue, migration, composition root and executable
`src/quinzee_cloud/modules/`	Service implementations, including deployments, managed engines and monitoring
`src/qhq/`	Native UI, background jobs and public-API client
`deps/simple/`	Build policy for the independent Simple package; no local toolkit source copy
`lib/boost_json/`	One compiled Boost.JSON implementation and wrapper
`infrastructure/engines/`	Fixed engine images, supervisor, TLS/CRL and engine telemetry
`infrastructure/functions/`	Registered-image runtime helper, private TLS gateway and real Functions host image
`tests/compatibility/service_bus/`, `tests/compatibility/functions/`	Locked official SDK/worker projects and isolated end-to-end fixtures
`infrastructure/directory/`	Dedicated PostgreSQL launcher, TLS slapd frontend and Windows attribute schema
`src/quinzee_cloud/modules/directory_protocols/`	Separate Entra/RBAC/managed-identity/LDAP frontends over the authority
`infrastructure/observability/`	Optional Collector/Grafana/Prometheus/Loki/Tempo configuration
`test/`	.NET/React compatibility application and its Bicep template
`tests/`	C++/helper/container/browser-adjacent compatibility checks and fixtures
`scripts/deps/`, `scripts/dev/`	Verified tool acquisition and development environment ownership
`deps/`	Dependency identities, patches and license evidence

Apply C++ standards (source checkout required), React standards (source checkout required) and the relevant cross-language standards before edits. Follow the public/private header boundary, RAII, explicit ownership and narrow third-party warning handling. Do not expose Boost transport types through `include/http/` or make QHQ depend on provider state.

The canonical directory map and prerequisites are in project overview (source checkout required). The documentation ownership map (source checkout required) determines where a changed fact belongs; avoid copying protocol tables into multiple manuals.

Related reading

- The Book of Quinzee: Meet Quinzee and its services
- The Book of Quinzee: Service compatibility and limits
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Scenario overview: AI Architecture Reviews
- Scenario overview: AI-Assisted Exploration of Quinzee Internals
- Scenario overview: Understanding How an Azure API Is Emulated

3. HTTP library and asynchronous execution

The transport library supports synchronous handlers and asynchronous completion. `server` owns an implementation with validated immutable configuration, a listener I/O context, connection registry, acceptor and lifecycle timers. Lifecycle calls from its own I/O thread are forbidden; a handler cannot safely destroy its server.

Public contract	Purpose
`request_handler` / `async_request_handler`	Immediate response or deferred `response_completion`
`request_executor`	Bounded worker/queue adapter with active, queued and rejected statistics
`request`, `response`	Owned protocol values, headers, cancellation and body access
`body_file`	File-backed body lifetime used for large streamed transfers
`server_config`, `tls_config`	Bounded transport options and TLS certificate configuration
`exchange_statistics` / `observe_exchange`	Optional completion accounting without Azure dependencies
`error`, `log_sink`	Lifecycle errors and optional thread-safe logging

The cloud composes eight request workers with a queue of 64. Saturation rejects work explicitly; it must not create unbounded threads or block the listener's I/O thread. Completion is delivered back to the appropriate connection executor. Cancellation and handler deadlines prevent abandoned connections from retaining work indefinitely; application code must still cooperate with cancellation.

Request/response validation rejects malformed headers and application-controlled framing. Keep-alive, HEAD semantics, chunking and socket writes remain transport responsibilities. Large body handlers use the file-backed API rather than converting every request into an in-memory string.

Read the public headers in include/http/ (source checkout required) and the application transfer design (source checkout required). `integration.http_stream` covers large-body lifetime, backpressure and completion accounting; ordinary HTTP/TLS tests cover connection and protocol failures.

Related reading

- The Book of Quinzee: Observe your application
- The Book of Quinzee: Troubleshoot a connection or workflow
- Quinzee Administrator’s Guide: Observe the platform and the application separately
- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: AI-Assisted Cloud Debugging
- Scenario overview: AI Architecture Reviews

4. Connection and streaming lifecycle

`lib/http/src/connection.hpp` defines the connection for plain and TLS streams. Asynchronous operations retain the connection through `shared_from_this()`. The registry holds weak ownership; final destruction notifies the server.

```
accept → [TLS handshake] → read headers/body → wait for handler → write response
                                 ↑                                      │
                                 └──────────── keep-alive ──────────────┘
                       error, cancellation or stop → close → destruction
```

The **waiting** state is important: worker execution is not a synchronous call inside the read callback. Completions are posted onto the connection's executor, with cancellation/deadline checks before a response is used. During graceful shutdown, active writes/waiting work get the configured shutdown allowance; stragglers are force-closed. Idle/partial reads and handshakes can be closed immediately.

Large request bodies spool to private unlinked files with bounded buffers. Responses can retain an opened immutable Blob generation, so replacement or deletion does not alter an in-flight reader. Block commit copies bounded chunks into a new generation and commits metadata atomically. Ordinary buffered request limits and large transfer limits are distinct.

A parsed exchange is observed once on completion or abandonment. Statistics measure from received headers through reads, worker waiting, execution and writes. Response wire bytes include HTTP framing but not TLS records. Idle connections and incomplete headers are outside this exchange accounting. Observer exceptions must not interrupt transport shutdown or response delivery.

See Blob transfer semantics (source checkout required), connection.hpp (source checkout required) and streaming integration tests (source checkout required).

Related reading

- The Book of Quinzee: Observe your application
- The Book of Quinzee: Troubleshoot a connection or workflow
- Quinzee Administrator's Guide: Observe the platform and the application separately
- Quinzee Administrator's Guide: Diagnose failures from the outside inward
- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: AI-Assisted Cloud Debugging

- Scenario overview: AI Architecture Reviews

5. TLS, identity and DNS

`lib/http/src/tls_context.cpp` configures a server context, checks the leaf/key pair and enforces the selected TLS minimum. Cloud operation is HTTPS-only. Native names and Azure-emulation names use separate listeners/certificates; this does not implement general SNI certificate selection or hot certificate reload on one listener. Server configuration errors fail startup rather than falling back to plaintext.

There are separate constrained development, Azure-emulation and engine trust domains. The telemetry tools have their own local CA. CA signing keys are not mounted into ordinary server/client containers. QHQ and SDK clients verify certificate chains and hostnames. The engine supervisor additionally serves a signed CRL for native clients that require revocation checking.

Identity issues RS256 client-credentials tokens with service audiences and a one-hour lifetime. Its signing key is process-local, so cloud restart invalidates existing tokens. Audience checks separate management, storage, vault and monitoring calls. Bootstrap credentials retain administrator access. New service principals, managed identities and interactive users require scoped ARM/Service Bus roles; other data planes deny them until their role families are implemented. See the identity contract (source checkout required).

The DNS module collects provider-published names and custom records, generates zones/CoreDNS configuration, and observes the Azure-emulation switch. Native engine records use actual instance IPs. Persisted DNS serials remain monotonic across rapid updates/restart; resetting a serial to wall time could otherwise leave CoreDNS serving old state.

References: certificate infrastructure (source checkout required), Azure emulation (source checkout required), DNS (source checkout required), engine TLS/CRL (source checkout required) and the user trust guide.

Related reading

- The Book of Quinzee: Certificates, DNS and Azure URLs
- The Book of Quinzee: Sign in and create a resource group
- Quinzee Administrator's Guide: Make names, certificates and permissions agree
- Quinzee Administrator's Guide: Operate the shared Entra and LDAP directory
- Scenario overview: Running Applications at the Edge
- Scenario overview: Building an Isolated Private Application Environment
- Scenario overview: Testing Security and Access Controls

6. Build System and Dependency Cache

CMake

- `CMakeLists.txt`: C++20, GCC 13+ check, `TRIGLAV_PROFILE`, output directories.
- `cmake/triglav_compiler_options.cmake`: warnings and per-profile flags; `triglav_apply_project_options()` and `triglav_apply_executable_options()` (static libstdc++/libgcc for debug and release).
- `cmake/triglav_dependencies.cmake`: runs `prepare.sh`, parses `name=prefix`, finds OpenSSL and Simple in CONFIG mode with `NO_DEFAULT_PATH`, and defines `triglav::boost_headers`.

Additional helpers: `triglav_apply_imported_options()` gives imported code (the Simple package wrapper, Boost.JSON) the profile flags and `-Wall -Wextra -Wpedantic` without the first-party baseline and `-Werror`. QHQ's post-build step copies the installed `SIMPLE_ASSET_DIR` and `src/qhq/assets/logos/` to `bin/<profile>/assets/`.

Parallel builds

The root Makefile adds `--jobs=$(JOBS)` to `MAKEFLAGS`, invokes CMake builds with a `+` recipe prefix so the generated makefiles join Make's jobserver, and runs CTest with `--parallel`. Never add a serial step; Docker builds use `-j"$(nproc)"`.

prepare.sh

```
for each manifest line (name version recipe source_identity url):
    identity = hash(recipe, package, archive sha, patches[, variant, toolchain, glibc, arch])
    lock .packages/locks/<name>.lock
    if installed/<platform>/<name>-<ver>-<id12>/.triglav-complete: reuse
    else: fetch (HTTPS archive + SHA-256, or Git + exact commit)
          → extract/checkout + patch (sources/, atomic)
          → build into staging (DESTDIR) → file hash list → identity
          → complete marker → rename into place
    print name=prefix
```

Adding a dependency means adding a manifest line and, if it needs compiling, a recipe function and an entry in `required_file_for`. Bump `recipe_version` when recipe behaviour changes, so existing entries are rebuilt.

`TRIGLAV_PACKAGES_DIR`, `TRIGLAV_DEPS_MANIFEST`, and `TRIGLAV_DEPS_PATCHES` redirect the script (the lifecycle test uses them). `TRIGLAV_DEPS_ALLOW_FILE_URLS=1` is for tests only. `--without-simple` skips the UI package for service-only builds. CMake uses this when `TRIGLAV_BUILD_QHQ=OFF`, as in the runtime Docker build.

Messaging and Functions dependencies

Service Bus adds source-built Apache Qpid Proton for AMQP framing/state and shares the verified SQLite dependency with Table Storage. The broker store remains in-process; it does not require an external broker container.

Functions uses an optional Microsoft Functions host image and locked .NET worker and binding packages. `make functions-example` publishes the worker and builds its local image; registration resolves the image to an immutable ID. Runtime image approval and helper operation remain separate from the native cloud build. See messaging and Functions for the source map.

Manual publication

`make pdf` builds the developer, administrator and internals books independently of the native targets. `make manuals-check` verifies their assembly, code, assets and destinations. `make manual-screenshots` refreshes the illustrations through an isolated live application test. The publishing guide (source checkout required) explains their caches, output paths and verification.

`make admin-manual-screenshots` captures the administrative atlas against a separate native HTTPS lab. The ordinary PDF build reads reviewed PNGs and starts no cloud service. Its pinned Node/Chromium tools live in `.packages`; all three PDF/HTML publications go into `bin/manuals`.

Related reading

- The Book of Quinzee: Stop, restart and clean up safely
- The Book of Quinzee: Build Quinzee from source
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Quinzee Administrator’s Guide: Configure the cloud and its launch environment
- Scenario overview: Working Without an Internet Connection
- Scenario overview: Building a Personal Cloud on a Home Server
- Scenario overview: Giving AI Coding Agents Their Own Cloud

7. Containers and runtime ownership

The Dockerfile uses a pinned Ubuntu builder and the local Triglav-OS `docker-dotnet:latest` runtime. C++ release/debug executables link their C++ and compiler runtimes statically while retaining the glibc runtime contract. The runtime stage contains binaries and license notices, not source or credentials. The .NET application is published separately into its own small runtime context.

The root Compose project runs cloud and CoreDNS with private state/leaf mounts, non-root ownership, dropped capabilities and loopback host publication. The Azure listener is reachable on private-network port 443. CoreDNS mounts only the generated DNS directory; it does not read the identity or secret stores.

`application-env.py` owns a separate application environment: unique test projects or a persistent development project, matching CA/DNS/client settings, the separate engine supervisor and cleanup constrained to owned resources. Client trust is mounted as public certificates, not installed into the host. The application image has a runtime; interactive SDK development uses another container as described in the user book.

Native engines are not child processes in the cloud container. Only the explicit supervisor has Docker-administrator access. Requests cannot choose arbitrary images, mounts, commands or Docker flags. Container/volume labels identify the supervisor's runtime and each instance for safe reconciliation and deletion.

The optional telemetry image starts only Collector, Grafana, Prometheus, Loki and Tempo through an explicit launcher. It is digest-pinned, drops capabilities, uses a read-only root and bounded resources, and exposes only loopback HTTPS ports. Grafana requires authentication. Health probes check the actual five services, not the upstream launcher's marker files. Tool data persists separately.

See container operations (source checkout required), application ownership (source checkout required), engine runtime (source checkout required) and telemetry operations (source checkout required).

Related reading

- The Book of Quinzee: Stop, restart and clean up safely
- The Book of Quinzee: Build Quinzee from source
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Quinzee Administrator’s Guide: Configure the cloud and its launch environment

- Scenario overview: Working Without an Internet Connection
- Scenario overview: Building a Personal Cloud on a Home Server
- Scenario overview: Giving AI Coding Agents Their Own Cloud

8. Testing and verification

Choose checks by the changed behavior. Build and CTest entry points are in AGENTS.md (source checkout required); the completion requirements remain mandatory. Documentation-only changes do not require rebuilding the application, but runnable examples must be checked against their actual SDK signatures and setup instructions.

Check	Coverage
`make test`	HTTP/TLS, host/modules, services, deployment expressions/runtime contracts, queues, DNS, monitoring, telemetry, QHQ and dependency lifecycle
`make PRESET=release test`	The same contracts with hardened optimized binaries
`make PRESET=asan test`	Memory/undefined behavior, on a compatible host or isolated builder
`make docker-tsan TSAN_ALLOW_ASLR_CHANGE=1`	Race checks with the explicitly authorized test-container policy described below
`make docker-lint`	C++ formatting and first-party Clang-Tidy diagnostics
`make docker-test`	Runtime images, HTTPS, constrained CAs, DNS/Azure names, persistence, JSON logs and official SDK calls
`make engines-test`	Real isolated native engines, TLS, lifecycle, persistence and owned deletion
`make service-bus-sdk-test`	Official .NET AMQP/TLS client, compiled Bicep, settlement, fan-out, SAS/bearer and unsupported operations
`make functions-test`	Real Functions host, HTTP key enforcement, Service Bus trigger, settings/lifecycle, helper recovery and deletion; requires the local lint runner image
`make manuals-check`, `make pdf`	Book assembly, links, assets, complete code inclusion and printable output
`make application-check`	Locked dependencies, .NET/React builds/style, Bicep, Playwright and automated accessibility
`make application-test`	Official .NET service clients, both SQL engines, Redis, queue recovery and cleanup
`make application-observability-test`	Real telemetry backends, cross-service correlation, source redaction and database/collector failures

Tests use private temporary state, ephemeral listeners or isolated Compose projects. Do not point destructive cleanup tests at a persistent developer instance. Application failures retain private logs/results; avoid publishing those directories without redaction.

Use `ctest --preset debug -R 'monitoring|telemetry'` for selected targets, or `build/debug/tests/monitoring_unit_test retention_and_durable_failure` for a single harness case. The exact registered targets are in tests/CMakeLists.txt (source checkout required).

ASan/UBSan and TSan are verified in containers where host-injected libraries interfere with sanitizer runtime startup. The TSan runner permits one exact personality argument inside its test container only, after explicit approval; it does not change host sysctls or use an unconfined filter. Read the TSan policy (source checkout required) before opting in.

The observability verification record (source checkout required) records the 23-target debug/release/sanitizer suites and SDK/backend checks. The application record (source checkout required) covers the browser and workflow contracts. These are dated evidence, not guarantees about untested architectures or every upstream SDK version. Manual assistive-technology checks remain distinct from automated axe/keyboard coverage.

The manual verification record (source checkout required) describes the C# tutorial checks, SDK-container setup and exporter sampling configuration.

Related reading

- The Book of Quinzee: Deliver work with Service Bus
- The Book of Quinzee: Service compatibility and limits
- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Scenario overview: Integration Testing Across Multiple Azure Services
- Scenario overview: Running Azure Integration Tests in CI/CD
- Scenario overview: Testing Application Upgrades and Regressions

9. Module host and service state

`module_host` composes independently described service modules. The catalogue registers identity, Resource Manager, deployments, Storage, Key Vault, App Configuration, Event Grid, DNS, PostgreSQL, MySQL, managed Redis, Log Analytics, Application Insights, Service Bus, Functions and App Service. Shared identity is exposed through separate Entra, RBAC, managed-identity and Windows LDAP frontend modules. `--modules` can restrict startup; dependency validation and lifecycle operations determine which combinations are usable.

A module declares routes and required/provided capabilities. Authentication, resource groups, provider inventory/lifecycle, DNS names, workspace access and service diagnostics cross module boundaries through capabilities. Shared state exposed by a capability must outlive its consumers; do not return a dangling pointer into a destructed module.

Request and lifecycle flow

1. The host establishes trace context, validates/splits routing inputs and chooses the best matching route.
2. Management routes authenticate through the identity capability. Data planes apply their own token/key/SAS rules.
3. An in-flight reference protects the selected module while its handler runs; host-table locks are not held across service work.
4. The provider validates version/schema, synchronizes its own state and emits the Azure-compatible response/error shape.
5. The host adds correlation/diagnostic accounting. Transport completion is observed separately after socket transfer.

Unload stops admission and waits for in-flight calls; dependent modules prevent unsafe removal. State is retained. Resource-group deletion is a separate provider lifecycle operation and cascades data removal, including native engine deletion. Active deployment work must be quiesced before that deletion can complete.

Persistence and data planes

Private per-module state replaces the old single state file. `document_file` provides atomic document updates; provider stores define their own rollback and schema rules. Blob generations/files, queue documents, secret versions, configuration revisions, operation records and telemetry rows have distinct limits and lifetime semantics. Only acknowledged durable changes should be reported as accepted when the API promises persistence.

Table Storage and the Event Grid delivery outbox use SQLite through the `sqlite_database` wrapper (WAL, `synchronous=FULL`, size caps, bound parameters only); a write or entity group transaction is one database transaction. Storage reports blob changes to Event Grid through the optional `event_publisher` capability, and Event Grid delivers to queues through the optional `storage_queues` capability, so neither module requires the other.

Queue receive/update/delete is serialized so competing consumers cannot claim the same visibility lease. Blob readers hold immutable generations. Key Vault and App Configuration values remain development data stored without an at-rest encryption service. Monitoring bounds persistence and rejects unsupported query syntax instead of inventing results.

Protocol references: host/API (source checkout required), modules (source checkout required), Storage (source checkout required), Queues (source checkout required), Tables (source checkout required), Event Grid (source checkout required), Key Vault (source checkout required), App Configuration (source checkout required), monitoring (source checkout required).

Directory, scoped permissions and workload identities

Identity owns the shared directory capability, backed by private JSON for legacy installations or dedicated PostgreSQL when configured. `entra_directory`, `azure_rbac`, `managed_identities` and `windows_ldap` are separate frontends over that authority. They can have independent admission/lifecycles without duplicating users, groups, credentials or role assignments. LDAP requires PostgreSQL and the optional socket/frontend processes; a loaded module alone is not an LDAP server.

Bootstrap credentials retain administrator status. New applications, principals and users start without resource permissions. The host guards supported ARM operations; Service Bus separately checks sending and receiving/settlement. App-role claims and resource RBAC are distinct: an API role does not grant ARM administration. Each data plane retains its own implemented authorization boundary.

Functions consumes the workload identity capability and exposes public identity attachments in ARM. Its runtime mailbox carries the private header; the helper injects it with the HTTPS App Service identity endpoint. App settings and public inventory omit the credential. The endpoint authenticates the workload and selects only an attached principal. System identity lifecycle follows its owner; user-assigned identity has an independent ARM resource.

The interactive login service keeps bounded, process-local form tickets, codes and rotating refresh grants. The HTML form uses a strict CSP with the validated callback origin allowed for its post-login redirect. Code redemption checks S256 PKCE, callback, client and scopes. Users' passwords use salted PBKDF2; ID tokens are explicitly separated from API access tokens. Read the identity reference (source checkout required) and implementation plan (source checkout required) before extending the contract.

The directory IPC chapter follows the exact PostgreSQL session, migration marker and slapd/Unix-socket path. The persistence chapter explains file replacement and embedded SQLite separately; there is no single shared database for all modules.

Related reading

- The Book of Quinzee: Store documents in Blob Storage
- The Book of Quinzee: Process work with Queue Storage
- Quinzee Administrator’s Guide: Administer storage, secrets and configuration
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Scenario overview: AI Architecture Reviews
- Scenario overview: AI-Assisted Exploration of Quinzee Internals
- Scenario overview: Understanding How an Azure API Is Emulated

10. QHQ and libsimple

libsimple

Simple is the independent widget toolkit extracted from Quinzee and TRIGLAV Setup. QHQ links its installed CMake target `simple::simple`; sources, assets and licences live in the dependency cache (build notes (source checkout required)). It renders into one ARGB surface and talks to the machine through a `platform` (display + input):

Backend	Use
`rasta`	Development: an SDL2 window (`bin/tools/rasta`) shows a shared framebuffer file and sends input over UDP
`framebuffer`	Native console: `/dev/fb0` + evdev
`offscreen`	Tests and `--screenshot`

Key pieces QHQ uses: `application` (event loop, timers, focus), `app_shell` (header, navigator, content host, footer), `navigator` in hierarchical mode, `panel`, `form`, `label`, `text_field`, `combo_box`, `list_view`, `button`, `stack`. Make toolkit changes in the Simple project, then update Quinzee's pinned revision. QHQ uses flat includes such as `simple/button.hpp`.

QHQ structure

File	Role
`main.cpp`	Options, platform, `application`, window; screenshot mode; signal flag
`qhq_window.*`	Shell, navigation catalogue, pages, controller actions
`qhq_provisioning.cpp`, `qhq_queues.cpp`, `qhq_observability.cpp`	Deployment/engine, queue and diagnostics/query pages
`qhq_service_bus.cpp`, `qhq_functions.cpp`	Messaging inventory and function-app lifecycle using public ARM APIs
`qhq_identity*.cpp`, `qhq_hosting_settings.cpp`	Graph identities/roles and hosting settings, public API mutations and private input handling
`tab_view.*`	Tab buttons over a `stack`
`background_jobs.*`	Worker threads with UI-thread delivery

File	Role
`cloud_client.*`	HTTP client for the emulator's public APIs
`assets/logos/quinzee_mark.svg`	Header brand mark

Threading model

```
UI thread                          worker (std::async)
  button → qhq_window action
    jobs_->run(work, done) ──────► cloud_client call (blocking, ≤10 s)
  timer (40 ms, only while            │
  jobs are pending) ◄──────────────── result
    done(result) → update widgets
```

Callbacks check that the client they used is still the current one, so a reconnect discards stale answers. `background_jobs` is destroyed before anything else in the window and waits for running work.

Adding a page

1. Add the service (and, if new, its category) to `navigation_catalogue()`.
2. Write `*_page()` returning a `page_column()` of `section()` panels; use `tab_view` if it has several parts, and register it in `tabs_`.
3. Register it in `build()` with `add(id, page)`.
4. Add `cloud_client` calls and a controller action that runs them as jobs.
5. Extend `integration.qhq` to drive the action and visit the page.

Visual checks

`qhq --screenshot` renders any page offscreen after the emulator has answered; `make qhq-screenshots` produces all of them. During F5 the Rasta framebuffer file `build/qhq/rasta.fb` holds the live frame (BGRA), which `ffmpeg -f rawvideo -pix_fmt bgra -video_size 1440x900 -i build/qhq/rasta.fb frame.png` converts.

The observability page uses authenticated public diagnostics and workspace-query APIs. It validates result shape before rendering, bounds displayed rows and discards replies from an earlier connection. Monitoring resource creation stays in ARM/Bicep; QHQ does not expose a separate workspace/component wizard.

Related reading

- The Book of Quinzee: Manage the environment with QHQ

- Quinzee Administrator’s Guide: Connect QHQ and make changes deliberately
- Quinzee Administrator’s Guide: Find every control in QHQ
- Scenario overview: Practising Azure Administration

11. Deployments and native engines

The deployment module preflights bounded inline ARM templates, resolves a supported expression/dependency subset, and schedules resource operations in dependency order. It dispatches through authenticated public provider routes; it does not write another provider's private state. Microsoft's pinned Bicep compiler remains an external tool that produces the ARM JSON input.

Deployment history and operation records survive restart. Secure parameter values and credentials do not become a replay log. Interrupted active deployments are explicitly failed on restart, while already-created resources remain for an incremental retry. Group deletion quiesces deployment work so a late task cannot recreate a supposedly deleted resource.

The managed-engine providers store desired state and submit schema-validated mailbox requests through `engine_runtime`. The separate Python supervisor reconciles pinned PostgreSQL, MySQL and Valkey containers, private volumes, certificates and native readiness. Cloud reads status and publishes actual engine IPs through DNS; resource existence alone is not readiness.

The supervisor owns CA keys outside the cloud data mount. Public CA/CRL material allows native clients to retain chain, hostname and revocation checks. Docker operations are limited to validated schemas and labelled owned resources; ARM input cannot supply arbitrary images or executable commands.

New desired revisions carry optional W3C context into the mailbox. The helper emits fixed operation spans/logs, plus a heartbeat independent of slow provisioning work. Diagnostic engine states are last-reconciled state; freshness comes from the heartbeat, not an assumption that a recorded Running instance must still be available.

The precise APIs, expression limits, deletion semantics, image identities and operator commands are in provisioning notes (source checkout required). Ownership/recovery checks are in `tests/engines/`; the ordinary CTest suite uses contract tests, while the native-engine/application targets exercise real protocol servers.

Related reading

- The Book of Quinzee: Deploy your application's resources with Bicep
- The Book of Quinzee: Turn infrastructure into repeatable deployments
- Quinzee Administrator's Guide: Control resources and infrastructure deployments
- Quinzee Administrator's Guide: Run PostgreSQL, MySQL and Valkey safely

- Scenario overview: Testing Bicep Deployments
- Scenario overview: Learning Infrastructure as Code with Bicep
- Scenario overview: Creating Reproducible Development Environments

12. Telemetry and Azure monitoring

Platform diagnostics and Azure application monitoring are separate paths. The platform's per-host telemetry service captures bounded request events, metrics and correlated JSON logs. The monitoring modules accept application telemetry into independently scoped, persistent workspaces. Enabling one does not imply that every event is copied into the other.

Platform path

W3C version-00 trace context enters at host dispatch and is propagated explicitly through deployment jobs, engine mailboxes and application queue messages. Thread-local request binding is non-owning and restored by RAII; it cannot be implicitly used as asynchronous context. The .NET consumer retains producer links for leased work recovered after restart.

Native request history and export queues are bounded. Metrics use fixed categories; URLs, credentials, payloads, SQL parameters and trace IDs do not become metric labels. Optional provider gauges include bounded sampled queue counts and explicit truncation. Transport metrics describe actual parsed HTTP exchanges, while handler metrics describe module calls and prepared body bytes.

The native OTLP/HTTP JSON adapter uses existing Boost/OpenSSL facilities. A worker performs verified HTTPS requests with cancellation/deadlines and bounded responses. Delivery is best effort with explicit failure/drop accounting. The private exchange coroutine's borrowed parameters are safe because the synchronous `post()` scope drains the I/O loop before destroying them; this narrow lifetime exception is documented at the declaration. The exporter is destroyed before captured host/worker providers.

The fixed C++ mapping is an OTLP producer, not a general OpenTelemetry SDK. The .NET application uses official OpenTelemetry/Azure Monitor libraries; the Python helper uses a small bounded fixed-schema producer. Interoperability is checked against a real Collector and persisted Prometheus/Loki/Tempo data.

Azure-compatible path

`log_analytics` owns workspace resources and bounded row storage. `application_insights` depends on that capability and owns components/ingestion. ARM IDs and generated customer/App/instrumentation IDs remain stable across redeployments. Component workspace bindings are immutable.

Ingestion normalizes the six supported envelope types, validates each item, batches atomic writes by workspace and reports partial acceptance/durable failure. The query evaluator implements only its documented, read-only pipeline; it has no external query engine or filesystem expressions. Component queries filter resource ID, workspace queries load only their workspace, and audiences separate ingestion/query/ARM. Full per-workspace RBAC is not implemented.

Use observability operations (source checkout required) for collector, TLS, retention and metric semantics; monitoring contracts (source checkout required) for APIs, limits, mappings and unsupported features; and the verification record (source checkout required) for failure-driven tests.

Related reading

- The Book of Quinzee: Observe your application
- The Book of Quinzee: Troubleshoot a connection or workflow
- Quinzee Administrator’s Guide: Observe the platform and the application separately
- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: AI-Assisted Cloud Debugging

13. The compatibility application

`test/backend/` is an ASP.NET Core 10 client of public APIs, with a React UI in `test/frontend/`. Its Bicep template provisions HTTP services, both SQL engines, Redis and monitoring resources. It imports official Azure SDKs and native protocol clients, not emulator implementation classes.

The workflow covers deployment/redeployment, secret/configuration access, Blob transfers/SAS, Queue receipt rotation and recovery, SQL/Redis writes, Application Insights export and Logs Query/Metrics. Each run has explicit steps, cancellation, bounded timeouts and a trace ID. The queue worker is a deterministic workflow step, not a continuously hosted service.

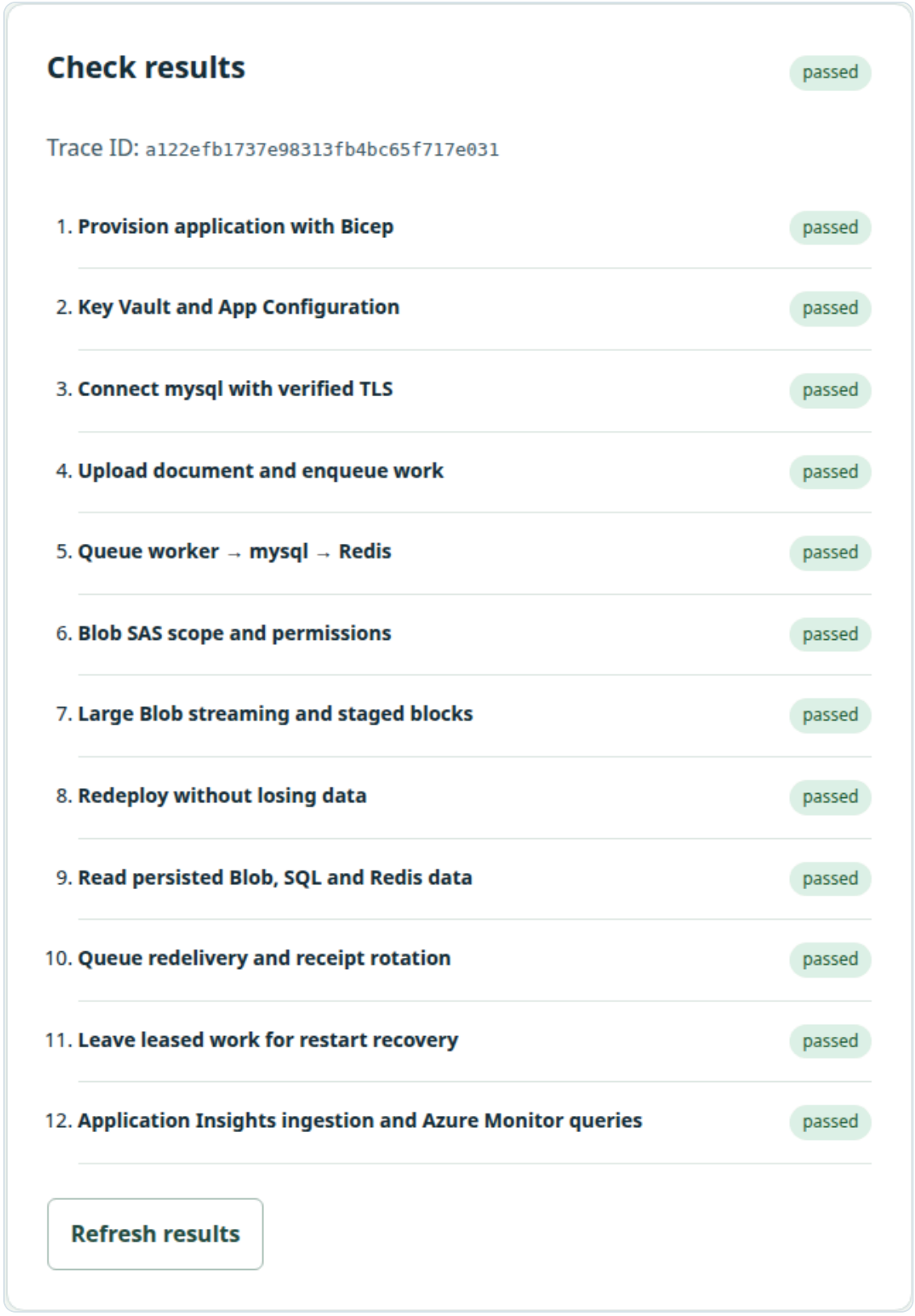
Settings, client secrets, database passwords and server leaf keys stay in private backend configuration. Browser access uses a separate local token exchanged for a Secure/HttpOnly/SameSite session cookie; mutations have origin/header checks. Errors report safe status/type information instead of SDK bodies/connection strings. Frontend state validation and escaped rendering are part of the compatibility test, not conveniences that should be removed for demonstrations.

`application-build.py` uses locked package graphs and prepares an image context from published artifacts/license notices. `application-env.py` owns runtime identity, CA/DNS setup, labels, subnet/ports and cleanup. Disposable tests use unique environments; persistent development uses `.local/application/development`. Attached clients must leave the network before the owning helper removes it.

`application-observability-test` verifies real backend correlation and source redaction, stops an owned PostgreSQL container to observe failure/recovery, and stops the collector while a complete workflow continues. It does not mutate unrelated engines. These tests establish the supported subset; they do not prove full Azure API parity or performance.

Read application operations (source checkout required), application acceptance (source checkout required), and the user C# tutorial. When updating SDKs, update locks and dependency/license evidence together, then run relevant browser and real environment checks.

The live workflow



The result panel from a disposable MySQL run includes real SDK operations, redeployment, queue recovery and Application Insights/Monitor compatibility.

Screenshot regeneration and PDF publication are described in manual publishing (source checkout required).

Related reading

- The Book of Quinzee: Meet Quinzee and its services
- The Book of Quinzee: Run your first application
- Quinzee Administrator’s Guide: Start with a small, understood installation
- Quinzee Administrator’s Guide: Connect QHQ and make changes deliberately
- Scenario overview: Developing for Azure Without Using Azure
- Scenario overview: Integration Testing Across Multiple Azure Services
- Scenario overview: Running Azure Integration Tests in CI/CD

14. Service Bus and Functions execution

Service Bus and Functions meet at the public messaging protocol, while keeping different ownership boundaries. The broker is native C++ inside `quinzee.cloud`. Function code executes in a separately operated real Functions host; the broker does not invoke application code itself.

Broker state and protocol work

`src/quinzee_cloud/modules/service_bus/` separates ARM resources, authentication, the SQLite store, AMQP codec/engine/management and the TLS listener. HTTP and AMQP operations share durable resource/message state. A store mutex serializes their access; one AMQP I/O worker serializes Proton events. Successful mutations are acknowledged after SQLite commit, with WAL and synchronous FULL enabled.

PeekLock ownership survives restart. Settlement checks the current entity and lock token; expired locks permit redelivery, and delivery limits send messages to the dead-letter subqueue. Topic publication atomically creates one copy per existing subscription. Resource limits and unsupported wire formats are enforced at the adapter/store boundaries, rather than trusting the SDK to avoid them.

The optional AMQP listener has its own port and module lifecycle. HTTPS readiness alone does not establish broker connectivity. SAS or bearer authentication is required for entity operations after AMQP negotiation. Private client trust in the SDK fixture retains hostname verification.

Desired state and real function hosts

`src/quinzee_cloud/modules/functions/` contains ARM routing, persistent state, runtime reconciliation and the HTTPS gateway. `infrastructure/functions/` owns image registration, the external Python runtime helper and container packaging. The cloud does not receive Docker privileges or run submitted code in-process.

The helper consumes validated desired-state mailboxes and executes only local images resolved through an operator registry to immutable IDs. Its private state is outside the cloud data directory. It issues per-app TLS material from a separate constrained CA and authenticates real host readiness before reporting Running. Status must match the current revision; stale status timestamps stop gateway forwarding. Upstream TLS verifies the app hostname even when the connection destination is the helper-selected container IP.

The public gateway forwards only bounded `/api/...` exchanges. Function-key enforcement belongs to the real host. Ordinary ARM inventory omits app settings and keys; authenticated dedicated APIs expose them. The helper owns containers, while explicit app lifecycle/deletion requests govern their execution/removal. Stopping the helper or unloading a module does not stop Service Bus triggers. Tombstones reserve deleted app names and retained private state requires explicit operator cleanup.

Changes and regression checks

Changed boundary	Relevant verification
Broker resource/auth/store semantics	`unit.service_bus` through CTest, including concurrent receive and persistence
AMQP adapter, TLS, lock settlement or SDK mapping	`make service-bus-sdk-test` with the locked official .NET client
Functions ARM/state/gateway or helper contract	Functions targets registered in `tests/CMakeLists.txt`
Runtime image, worker bindings or reconciliation	`make functions-test` with the real host and registered local image
QHQ inventory/lifecycle	`integration.qhq`, using public APIs

The Service Bus fixture locks `Azure.Messaging.ServiceBus`; the Functions project locks both worker packages and generated binding-extension dependencies. Keep both Functions locks in sync when changing bindings. The full fixture checks HTTP key rejection/success and automatic queue settlement, not merely container startup. See the testing chapter for runner prerequisites and profile/security constraints.

The full contracts and limits remain in Service Bus (source checkout required) and Functions (source checkout required). Design rationale is recorded in decision 025 (source checkout required) and decision 026 (source checkout required). The corresponding user walkthroughs explain provisioning, credentials, invocation and safe shutdown.

App Service web apps share the Microsoft.Web provider and external runtime mailbox, with independent feature admission through `web.hosting`. Their image entrypoint, native hostname, readiness probe and constrained CA differ from Functions. See App Service architecture and verification (source checkout required) and the implementation record (source checkout required).

The runtime reconciliation chapter follows the Unix file handoff, helper locks, Docker ownership, mounts, cgroups, identity credentials and forced container replacement. Those details explain why stopping the gateway is different from stopping a queue-triggered workload.

Related reading

- The Book of Quinzee: Run HTTP and Service Bus functions
- The Book of Quinzee: Host a web application
- Quinzee Administrator’s Guide: Operate Functions and App Service workloads
- Quinzee Administrator’s Guide: Keep events and messages moving
- Scenario overview: Developing an Azure Functions Application Locally
- Scenario overview: Experimenting with Cloud Architecture
- Scenario overview: Using Quinzee as an Internal Server for a Small Office

15. From a shell command to a running service

At the Unix level, Quinzee is a set of ordinary processes. A module is a C++ object inside the cloud process; it is not a daemon, container or dynamically loaded shared library. A PostgreSQL server, by contrast, really is another process with its own address space, sockets and storage. Begin an investigation by identifying the boundary that owns the work.

Follow startup in order

Consider a native launch with an existing private configuration:

```
bin/debug/quinzee.cloud --config /absolute/path/to/cloud.json
```

The shell starts the executable. The kernel maps the executable and its required runtime libraries, supplies arguments/environment and inherited file descriptors, and begins execution under the shell's UID and GID. Quinzee does not become root or install a service manager as part of this command. Relative paths resolve against this process's current working directory.

The composition root in `src/quinzee_cloud/main.cpp` then performs these steps:

1. Parse the JSON configuration and command-line overrides. Invalid settings fail before serving requests; the CLI overrides the file.
2. Block SIGINT and SIGTERM before creating worker threads. Those threads inherit the blocked signal mask.
3. Prepare diagnostics, migrate the old single-file format when applicable, construct the module host and load the selected modules and dependencies.
4. Create the request executor with eight workers and a queue of 64 requests.
5. Construct the native HTTPS listener and optional Azure-name listener, loading their certificate/key pairs. They share the service executor and module host.
6. Start the listeners and wait for a termination signal using `sigwait`.

Some modules start additional work while loading: the AMQP listener, deployment processing, Event Grid delivery or optional directory socket. Consequently, the process can have more threads and sockets than the two HTTP listeners suggest. The exact count depends on the selected modules and enabled facilities.

Distinguish processes from threads

```
operator shell / development launcher
  +-- quinzee.cloud
  |     +-- main thread: sigwait
  |     +-- HTTPS listener I/O thread(s)
  |     +-- eight request workers
  |     +-- module-specific workers and optional telemetry export
  +-- qhq: UI loop and asynchronous client jobs
  +-- rasta: viewer process, shared framebuffer and UDP input
  +-- engine helper: Python controller, Docker CLI subprocesses
  +-- hosting helper: Python controller, Docker CLI subprocesses
  +-- slapd: optional LDAP frontend (ldap.py execs this executable)

Docker daemon / container runtime
  +-- cloud and CoreDNS containers when using Compose
  +-- managed PostgreSQL / MySQL / Valkey containers
  +-- Functions host + isolated .NET worker, or web application containers
```

This is an ownership sketch, not a promise about every PPID on every system. Container processes are supervised by the container runtime; they are not Unix children of the cloud just because an ARM request caused them to be created. Stopping a Python controller therefore does not automatically stop its containers.

Threads share process memory and file descriptors. Mutexes protect C++ state; they do not protect a directory against a separate process editing its files. The request workers wait on a condition variable when idle. In Linux tools that can appear as a futex wait. A sleeping worker is usually idle, not deadlocked.

Observe one known native process

Choose the PID of your own lab instance, rather than terminating every process with a matching name. The commands below use an explicit PID:

```
cloud_pid=12345
ps -p "$cloud_pid" -o pid,ppid,user,stat,etimes,comm
ps -L -p "$cloud_pid" -o pid,tid,stat,pcpu,wchan:28,comm
readlink "/proc/$cloud_pid/exe"
readlink "/proc/$cloud_pid/cwd"
cat "/proc/$cloud_pid/limits"
```

Replace 12345 before running. Expect one PID with several TIDs. The executable and working directory tell you which build and relative paths are in use. The limits file explains ceilings such as open files; it does not mean Quinzee can safely use every descriptor allowed by the OS. Its own queues, connections and resource stores have additional bounds.

For a container, obtain its host PID with a narrow inspection:

```
cloud_container=$(docker compose ps -q quinzee-cloud)
docker inspect --format '{{.State.Pid}} {{.State.Status}}' "$cloud_container"
```

That is a host PID, which can differ from the PID visible inside the container. With a remote Docker context it belongs to the remote Docker host, not your workstation's `/proc`.

Understand graceful termination

SIGINT from Ctrl+C and SIGTERM from a supervisor wake the main thread's `sigwait`. Normal C++ code stops HTTP listeners; it does not perform complex cleanup from an asynchronous signal handler. Each listener stops accepting, closes idle/partial connections, and gives waiting work or active writes the configured shutdown allowance before closing stragglers. The library default is five seconds per listener.

Leaving `main` then destroys local owners in reverse order: the exporter, listeners, request executor and module host are released while their dependencies still exist. The executor requests worker stop, wakes sleepers and joins threads. Joining cannot forcibly interrupt arbitrary blocking application code; handlers and external calls must honour cancellation or have their own bounded timeouts. The earlier “shutdown complete” log does not mean every local destructor has already finished.

SIGKILL, an OOM kill or power loss skips this C++ cleanup. A container manager may escalate to SIGKILL when its grace period expires. Do not treat that timeout as a transaction coordinator: recovery still depends on the individual durable stores. The persistence chapter explains which operations survive interruption.

Closing QHQ only closes a client. Unloading a module changes admission and worker lifetime inside the cloud. Stopping a hosted app changes an external workload. These are three different operations with different owners.

Related reading

- The Book of Quinzee: Stop, restart and clean up safely
- The Book of Quinzee: Build Quinzee from source
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Quinzee Administrator’s Guide: Configure the cloud and its launch environment
- Scenario overview: Working Without an Internet Connection
- Scenario overview: Building a Personal Cloud on a Home Server
- Scenario overview: Giving AI Coding Agents Their Own Cloud

16. Follow a packet through names, sockets and namespaces

A hostname is not a listening socket. DNS maps a name to an address; a TCP port selects a listener at that address; TLS authenticates the server name; HTTP host and path select the service. A failure at one layer cannot be fixed by changing an unrelated layer.

Trace a native HTTPS request

For `https://localhost:11443/health`, the client resolves localhost and opens a TCP connection. The server's listening descriptor accepts a new socket. OpenSSL performs TLS through Asio; Beast parses decrypted HTTP bytes. The listener's I/O thread advances connection state and dispatches service work to the executor. The completion returns to that connection's executor for serialization and socket writes. Keep-alive can reuse the accepted socket for another request.

```
client resolver -> address:port -> TCP accepted socket -> TLS
   -> HTTP parse -> bounded request queue -> module handler
   -> completion on connection executor -> TLS records -> client
```

Nonblocking socket I/O means a thread need not wait for every byte. It does not make every service operation nonblocking: JSON persistence, SQLite commits and some outbound calls occupy a service worker while they execute. When the 64-item queue is full, the executor returns 503 rather than creating more workers. Increasing a Unix file-descriptor limit does not remove that queue bound.

Understand the standard Docker network

Compose creates a private bridge network with cloud address `172.29.53.10` and CoreDNS `172.29.53.53` by default. Each container has its own network namespace, interfaces and loopback. Docker connects it to the bridge and provides host-port publication. The exact firewall/proxy mechanism depends on Docker's setup; the important observable mapping is host loopback 11443 to cloud-container 11443.

Inside an application container, `127.0.0.1` means that application container. It does not mean the operator host or Quinzee Cloud. Joining the bridge also does not select CoreDNS: the client must use the configured DNS server.

Connection	Standard destination
Host to native cloud HTTPS through Compose	localhost:11443, published by Docker
Lab container to native cloud APIs	quinzee.cloud:11443 on the private network
Lab container to Azure-name HTTPS	Supported Azure hostname on private port 443
Host DNS diagnostic	localhost:10053, UDP or TCP publication to CoreDNS:53
Service Bus AMQP	Namespace hostname, TLS port 5671 when enabled
Native SQL/cache client	Engine container's published DNS address, ports 5432/3306/10000
Cloud gateway to Functions/web runtime	Selected workload IP, verified hostname, TLS port 8443

The cloud does not proxy SQL or Redis protocol bytes through its HTTPS listener. Provisioning uses ARM; application queries go to the real engine.

Inspect each layer independently

These host commands assume the standard Compose lab already exists. `ss` comes from iproute2, `dig` from a DNS client package, and the TLS probe uses OpenSSL:

```
ss -ltn
ss -lun
docker compose port quinzee-cloud 11443
dig @127.0.0.1 -p 10053 quinzee.cloud A +short
openssl s_client -connect localhost:11443 -servername localhost \
  -verify_hostname localhost -verify_return_error \
  -CAfile .local/certificates/ca/quinzee-dev-root-ca.crt </dev/null
curl --fail --silent --show-error \
  --cacert .local/certificates/ca/quinzee-dev-root-ca.crt \
  https://localhost:11443/health
```

A correct A record with a refused TCP connection points toward binding, routing, publication or process state. A successful TCP connection with failed TLS points toward certificate name, chain or trust. Successful TLS with 401/403 means you have reached an authentication/authorization boundary. Do not skip those tests by disabling certificate checks.

Distinguish TLS names from network destinations

HTTP native and Azure-name emulation use separate listener/certificate contexts. The host's certificate must cover the name the client verifies. Changing the HTTP Host header does not change the certificate already received during TLS. The Service Bus AMQP TLS adapter

separately supports its configured native and Azure namespace certificates; it is not the HTTP listener's certificate switch.

For an upstream hosted app, the gateway connects to the helper-selected IP but still verifies the app hostname and its constrained runtime CA. Connecting to an IP is therefore compatible with strict name validation when the client deliberately carries the correct TLS server name. It is not permission to turn verification off.

Look inside the container boundary without dumping secrets

```
cloud_container=$(docker compose ps -q quinzee-cloud)
docker inspect --format '{{json .NetworkSettings.Networks}}' "$cloud_container"
docker inspect --format '{{.HostConfig.ReadonlyRootfs}} {{json .HostConfig.CapDrop}}' \
  "$cloud_container"
```

Network namespaces isolate addresses; mount namespaces isolate filesystem views; cgroups bound workload resources; UIDs and filesystem modes govern access to shared files. These mechanisms cooperate but do not substitute for one another. A read-only container root still permits writes into an explicitly writable bind mount. A host bind mount refers to host inodes; it is not automatically a private copy. Docker/user-namespace mappings can alter how numeric IDs appear, so check the actual host and container ownership when diagnosing access failures.

The cloud uses no Docker socket. CoreDNS sees the generated zone directory only. The standard cloud runs without Linux capabilities and uses a container-local unprivileged-port setting for port 443. CoreDNS has its separately documented port-53 setup. Neither configuration requires making the whole host's ports unprivileged.

Related reading

- The Book of Quinzee: Certificates, DNS and Azure URLs
- Quinzee Administrator’s Guide: Make names, certificates and permissions agree
- Scenario overview: Running Applications at the Edge
- Scenario overview: Building an Isolated Private Application Environment

17. Files, transactions and what a crash can leave behind

An API response, a C++ object and bytes on disk are different kinds of state. Quinzee uses several persistence mechanisms because an uploaded Blob, a message lease and a directory permission change have different access patterns. There is no single database transaction spanning all services or a whole ARM deployment.

Follow an atomic JSON replacement

The shared `document_file` helper writes the complete next document to a sibling `.tmp` path, using an owner-private creation mode by default. It handles short writes and interrupted writes, flushes the file with `fsync`, closes it and renames it onto the final path. It then attempts to flush the parent directory.

```
old state.json remains readable
  -> open state.json.tmp -> write all bytes -> fsync -> close
  -> rename temporary to state.json -> attempt parent-directory fsync
new opens see the new document
```

A same-filesystem rename switches the directory entry atomically. A reader does not see half of the new JSON merely because the writer needed several writes. The file flush and directory flush address durability at different levels: the first concerns contents; the second concerns the name-to-inode update.

There are implementation limits worth knowing. This helper uses a fixed sibling temporary name and expects its owner to serialize writes. It is not an interprocess locking service. Its parent-directory open/fsync is best effort: those failures are not reported as strictly as the temporary-file flush failure. It is therefore incorrect to describe every document update as an unconditional power-loss guarantee on every filesystem. Recovery and backups remain necessary.

The private-directory helper applies restrictive permissions when creating a directory but does not generally repair an already existing permissive directory. Likewise, a creation mode is not a recursive chmod of existing state. Specific boundaries such as the directory connection file and LDAP socket parent perform their own stricter checks. Keep the entire state tree owned and protected by the service account; do not rely on a filename extension to protect secrets.

Understand an open, deleted upload file

Large Blob uploads spool through `body_file`. It creates a mode-0600 temporary file with `mkostemp` and close-on-exec, then immediately unlinks its name. The open file descriptor still refers to the inode. `pwrite` appends bounded chunks; `pread` reads by offset without changing a shared seek position.

This explains an apparent Unix puzzle: the uploads directory can look empty while disk space is in use. The file has no directory name but is still open. The kernel reclaims it after the last descriptor/reference closes. A process exit also closes its descriptors. `O_CLOEXEC` prevents accidental inheritance across a later exec; it is not a substitute for normal RAII closure.

Blob publication copies bounded chunks into a named generation and commits metadata. Readers keep an opened immutable generation, so replacing or deleting the Blob name does not change bytes already being downloaded. An old reader can finish while a new reader sees the replacement. This is intentional lifetime management, not evidence that deletion failed.

Separate embedded SQLite from PostgreSQL

Table Storage, Event Grid's delivery outbox and Service Bus use embedded SQLite. SQLite runs as library code inside the cloud process: there is no sqlite daemon, TCP listener or external database container for those stores.

The wrapper opens private files without following a final symlink and enables WAL, synchronous FULL, foreign keys and an untrusted schema setting. It uses a busy timeout, bound SQL parameters and a maximum page count. Provider code adds its own record/message/size limits and synchronization.

In WAL mode, a committed transaction may reside in the `-wal` file before a checkpoint copies its pages into the main database. The `-shm` file supports shared coordination. Copying the main `.db` alone while the service is live can lose committed changes. Do not delete WAL files to “unlock” a running store. Use a supported database backup method or a stopped, consistent instance copy.

One Table entity-group transaction either commits its changes or rolls them back. One Service Bus topic publication commits the subscription copies together. Neither transaction includes arbitrary updates to another provider or external engine. PostgreSQL-backed identity is a separate connection/session and is explained in the next chapter.

Inspect metadata, not private contents

For the standard stopped or running local instance, these commands inspect ownership and sizes without printing credentials or document bodies:

```
stat -c '%a %U:%G %n' .local/quinzee .local/quinzee/credentials.json
du -sh .local/quinzee
cloud_pid=12345
ls -l "/proc/$cloud_pid/fd"
cat "/proc/$cloud_pid/io"
```

Choose your actual lab PID. Open descriptors may show sockets, databases or `(deleted)` payload files. Do not read through `/proc/<pid>/fd/N`: that can expose data and can interact with a descriptor's underlying object. `/proc/.../io` reports process-level counters, not a per-service audit or proof that storage hardware has honoured every flush.

Reason about interruption at the right boundary

Interrupted operation	What to investigate after restart
JSON replacement before rename	Old committed document; a leftover temporary file may exist
Blob upload before publication	No completed object from that request; unlinked spool descriptors close on exit
Blob replacement with an active reader	Old opened generation can outlive the name change
SQLite transaction	SQLite recovery and the provider's acknowledged transaction boundary
ARM deployment after some resources succeeded	Failed deployment history and retained completed resources; no universal rollback
External SQL write	Native engine transaction/WAL or redo recovery, independent of cloud metadata
Directory migration	PostgreSQL contents plus local destination marker and the retained legacy input

Avoid editing live JSON or SQL to force a desired status. Those files are private implementation state; changing them bypasses locks, validation, revisions and ownership checks. A useful recovery drill restores matching state and dependencies and then exercises public APIs, rather than merely proving that an archive opens.

Related reading

- The Book of Quinzee: Store documents in Blob Storage

- The Book of Quinzee: Process work with Queue Storage
- Quinzee Administrator’s Guide: Administer storage, secrets and configuration
- Quinzee Administrator’s Guide: Keep events and messages moving
- Scenario overview: Experimenting with Cloud Architecture
- Scenario overview: Testing Backup and Recovery

18. One directory behind HTTPS and LDAP

Entra-style HTTPS and LDAP are two entrances to one directory. The important Unix question is which process parses each protocol, where the authoritative bytes live, and which permissions allow the processes to communicate.

Follow the actual process path

```
Graph/OAuth client -- HTTPS --> quinzee.cloud
                                    |
LDAP client -- LDAPS --> slapd -- private Unix stream socket
                                    |
                              shared directory objects
                                    |
                          libpq connection and writer lease
                                    |
                        dedicated PostgreSQL database
```

OpenLDAP terminates LDAP TLS, parses LDAPv3 and applies its frontend limits. Its `back-sock` backend forwards supported operations over a local Unix socket. The C++ backend implements the shared directory view and authentication. Slapd does not hold another writable copy of the users in an MDB database here. There is no background LDAP-to-Graph synchronization job.

The application-facing PostgreSQL ARM module is a different service. Its databases are application resources and can be deleted with their groups. The directory's dedicated PostgreSQL dependency must remain available before the cloud can administer those resources.

Understand the PostgreSQL session

`directory_store` reads a private regular JSON connection file of at most 16 KiB. It passes separate values to `PQconnectdbParams`, without URI expansion. TCP requires `verify-full` and a root certificate. An absolute socket directory in the host field selects PostgreSQL's local Unix socket transport; TLS is disabled for that transport, so filesystem and PostgreSQL authentication rules matter.

The connection is named `quinzee-directory`. It sets a five-second connect timeout, statement timeout and lock timeout. Queries use nonblocking libpq calls with `poll` on its socket and an approximately six-second client deadline. This is bounded synchronous work from the caller's

perspective: it can occupy a cloud worker or the LDAP backend worker while waiting for PostgreSQL.

The schema is deliberately small: `public.quinzee_directory` stores a JSONB snapshot keyed by tenant UUID. The directory layer limits its serialized state to 8 MiB and each object collection to 1,024 records; the SQL column also has a 16 MiB text-size guard. It is not a normalized SQL table per user or group.

At startup the connection obtains a session-level PostgreSQL advisory lock for the tenant. A second writer is refused. A mutation validates a candidate snapshot, commits its single parameterized upsert, then publishes the new in-memory state. This ordering keeps a failed write from becoming the accepted in-memory permission state. Other directory entry points check backend health.

If the session fails, the store drops the connection and denies further access. It does not silently reconnect: reconnection would need to re-establish writer ownership and trustworthy state. Restore PostgreSQL and restart the cloud. An OS file lock and a PostgreSQL advisory lock are different mechanisms; this lease belongs to the database session, not a lock file on the operator host.

Follow migration and restart

Without a database configuration, a legacy installation uses its private `modules/identity/directory.json`. When PostgreSQL is configured, the first successful import retains IDs and credentials and writes `directory-postgresql.json` with a destination fingerprint. Existing PostgreSQL state is reused instead of overwriting it with old JSON.

The destination fingerprint binds host, port, database and user. Removing the database configuration, changing that destination or losing a previously migrated tenant row fails rather than reviving stale permissions. The old JSON is retained as migration input/history, not continuously updated as a standby database. Restore the PostgreSQL snapshot and matching local state together.

Persistent directory records are distinct from authentication session state. The OAuth signing key is generated in memory when the authority loads; a cloud restart changes it and invalidates tokens at the restarted authority. Login tickets, authorization/device codes and rotating refresh grants are also process-local. A persistent user therefore still needs to sign in again after a restart. External APIs caching old signing keys have their own token-validation and expiry behavior; the cloud cannot recall a token from their memory.

Inspect the Unix socket boundary

The cloud binds the configured absolute socket path, limited to 100 bytes, in a parent directory with no group/other permissions. The socket itself is mode 0600. The implementation relies on this filesystem boundary and the trusted frontend running as the

same UID; it is not a separate SO_PEERCRED-based identity protocol. A process with that UID is inside the local trust boundary.

```
stat -c '%F %a %U:%G %n' .local/directory-run \
  .local/directory-run/directory.sock
ss -xl
```

Use your actual configured paths. A socket pathname alone does not prove that a listener is alive. The cloud refuses to unlink a pre-existing path during startup because it might belong to another live process. Normal destruction stops its worker and removes its socket; an unclean exit can leave a stale path. Verify ownership and absence of a live listener before any manual cleanup.

The Asio backend permits at most 16 active sessions, reads a bounded 16 KiB request until the back-sock blank-line delimiter, writes one response and closes the connection. Each session has a five-second timer. Synchronous backend work can delay that timer's callback, so it is not a hard preemption guarantee. The implementation also bounds results and directory queries; the public API cannot ask the kernel to kill an in-progress C++ call safely.

Understand what slapd contributes

`infrastructure/directory/ldap.py` writes a private `slapd.conf` and calls `os.execv`. Exec replaces the Python process image with slapd while retaining its process identity. There is no Python reverse proxy left forwarding bytes. The generated configuration includes core/cosine/inetOrgPerson/NIS schemas and Quinzee's bounded Windows attribute schema, selects `database sock`, and points to the private backend socket.

The frontend listens on LDAPS only, defaults to localhost:11636, requires TLS 1.2 or later and a security strength factor of at least 128, uses four threads, and sets size/time/idle/write limits. Back-sock conveys the bound DN and SSF; the backend checks enabled identities before serving reads. Anonymous users may authenticate but cannot enumerate the directory. Bind credentials cross the private socket, so never publish that socket or trace its payload.

POSIX users/groups and Windows-oriented users/groups are projections of the same objects. Direct members become memberUid or member/memberOf values in the appropriate view. Stable POSIX IDs and generated account names are directory data; they are not entries automatically installed in `/etc/passwd` or `/etc/group`. App-role grants use direct user membership, while resource RBAC has its separate bounded transitive membership rules.

No PAM, NSS or SSSD configuration is changed by the server. Linux login requires an explicitly configured client. No Kerberos KDC, Windows domain join, SID-based security descriptor engine or AD replication service appears merely because the LDAP view exposes Microsoft-shaped attributes.

Related reading

- The Book of Quinzee: Sign in and create a resource group
- The Book of Quinzee: Give applications and people the right identity
- Quinzee Administrator’s Guide: Operate the shared Entra and LDAP directory
- Scenario overview: Testing Security and Access Controls

19. From desired state to a real container

An ARM PUT cannot directly become a shell command. Quinzee first validates and persists a resource model. A separate controller translates a bounded desired state into Docker operations. This separation is both an implementation boundary and the reason an ARM resource can exist while its runtime is not ready.

Follow the handoff through files

```
ARM caller -> provider -> private resource record
                        -> requests/<instance>.json
operator-run helper -> validate schema/revision -> Docker CLI -> Docker daemon
                    -> probe real protocol/TLS
                    -> status/<instance>.json
provider -> accept matching revision/status -> API and DNS/gateway readiness
```

Engine mailboxes live under `<data-dir>/engine-runtime`; hosting mailboxes under `<data-dir>/functions-runtime`, including App Service. These are files shared through the operator's selected paths or mounts, not network message queues. Their credential-bearing contents stay private. The helpers publish documents through temporary-file flush/replace operations so readers see complete records.

A revision identifies the desired incarnation of a resource. If an old probe finishes after a settings change, its success must not certify the new revision. The provider compares status to current desired state and applies freshness checks. Engine diagnostics also have a separate two-second heartbeat writer, so a slow operation and a dead supervisor can be distinguished. Hosting reports timestamps with reconciled status; do not assume it uses the same heartbeat file. A Running hosting status older than 15 seconds or more than five seconds in the future loses its routing address. An outstanding change has a 180-second provisioning deadline. These are current implementation bounds, not a distributed clock-synchronization guarantee.

Understand controller ownership

The engine helper derives an owner identifier from the resolved mailbox path. Its default private state is outside the cloud data mount. Both runtime helpers take a nonblocking exclusive `flock` on their private helper lock file. The lock is held by an open file description and released on process exit; the continued existence of the lock filename does not prove that the lock is held.

Docker ownership labels identify the helper and resources it may reconcile. The helper inspects labels before adopting or removing objects. Do not start a second controller with a different private-state directory against the same mailboxes, or rewrite labels to bypass a mismatch. Moving paths can change owner hashes and the objects a helper expects to find.

The Python process invokes the Docker CLI with structured argument lists. The CLI talks to the Docker daemon selected by the operator's context. With a remote daemon, bind-mount source paths refer to that daemon's filesystem, which may not be the controller's filesystem. The standard lab assumes the matching local Docker host. Access to the daemon is a privileged operator capability even when the Python process itself is non-root.

Native engine creation, step by step

1. Validate the fixed schema, supported engine kind, names, image identity and desired action. ARM input cannot choose arbitrary Docker flags or executables.
2. Establish ownership-labelled volume/container state. Prepare engine-specific configuration, credentials and TLS material outside the cloud mount.
3. Run the pinned PostgreSQL, MySQL or Valkey image with its prescribed user, network, mounts and limits: 1 GiB memory, two CPUs and 256 processes.
4. Probe the native protocol over verified TLS. Creating a container is not a successful database query.
5. Record status and actual network address. The provider's DNS contribution points clients at that engine address.

Data lives in engine volumes, not solely in ARM JSON. PostgreSQL has its own WAL, MySQL its own engine recovery, and Valkey its configured append-only persistence. The engine helper also owns a CA and a bridge-reachable CRL responder. Stopping the helper can leave databases alive while removing that responder and stopping reconciliation/heartbeat updates.

The helper's stop action stops an owned SQL container while preserving volume data. Delete is different: it can remove the owned volume and private runtime credentials. Resource-group deletion can initiate that sequence. It is not a general container backup or volume-adoption API.

Functions and web-host creation, step by step

The hosting registry maps an operator-approved image name to an immutable local image ID. The helper uses `--pull=never`; provisioning cannot fetch arbitrary code from a registry. Functions overrides the image entrypoint to start the real Functions host. App Service uses the approved image's own entrypoint.

The workload runs as the helper's numeric UID/GID, with a read-only root, dropped capabilities, no-new-privileges, and fixed CPU/memory/process limits. Writable tmpfs mounts provide `/tmp` and `/home/LogFiles`; their content is ephemeral. Private generated files supply TLS, trust, settings and host secrets. An ordinary application gets no Docker socket or cloud bootstrap credentials.

The helper injects settings through a private environment file. Those values become process environment inside the container; they are not magically hidden from the Docker administrator. Avoid unrestricted `docker inspect` or `/proc/<pid>/environ` dumps when investigating a failure.

Functions readiness uses a verified host-status request with the private master key. Web readiness checks `/` over TLS on 8443. Once ready, the cloud gateway makes its own verified upstream connection using the correct constrained CA. Functions admits bounded `/api/...` traffic and leaves function-key enforcement to the real host. Web hosting forwards normal paths and leaves application login to the application.

A managed identity attachment causes the helper to inject a private workload header and HTTPS token endpoint. The endpoint checks the header and the actual site attachments, then issues a token for the selected principal. The header is a credential, not an Azure resource ID. Identity attachment alone grants no resource access; authorization still depends on assignments.

Separate revision replacement from graceful application draining

The hosting helper removes an existing owned container when the revision changes, the action is not start, or the image is cleared. This path uses forced container removal; it does not promise application-level request draining. Settings changes can therefore interrupt in-flight work. Store durable application data in a real service and design message consumers for retries.

Stopping the hosting helper leaves existing containers alone. Unloading App Service blocks its gateway but can leave its process running. A Functions Service Bus trigger can keep consuming independently of the public HTTP gateway. To stop execution, submit the supported app Stop operation while its helper is running and wait for the resulting status.

Observation	Meaning to check
Resource exists, no container	Missing helper, wrong paths, rejected request or unapproved image
Container running, resource not ready	Protocol probe, TLS, credentials, revision or timestamp
API stopped, messages still consumed	Gateway/module stopped but workload container still alive
New helper cannot adopt an old container	Ownership/path/private-state mismatch

Observation	Meaning to check
Data missing after Delete	Destructive resource lifecycle, not a controller restart

This model explains why backups need cloud state, dedicated directory data, engine volumes and helper private state together. Copying only the ARM resource record cannot recreate the contents of an external database.

Related reading

- The Book of Quinzee: Run HTTP and Service Bus functions
- The Book of Quinzee: Host a web application
- Quinzee Administrator’s Guide: Operate Functions and App Service workloads
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Scenario overview: Developing an Azure Functions Application Locally
- Scenario overview: Working Without an Internet Connection
- Scenario overview: Using Quinzee as an Internal Server for a Small Office

20. Inspect a disposable instance from Linux

This exercise starts a separate native cloud, observes its Unix boundaries and stops only that instance. It uses no Docker services and changes no host DNS or trust store. Complete a native debug build first. You need Bash, Python 3, OpenSSL, curl and ordinary Linux procps/coreutils tools. `ss`, `strace` and `inotifywait` are optional diagnostics; absence of a tool is not a failed service.

Step 1: start a private HTTPS lab

Run the following in a dedicated Bash terminal from the repository root. Keep this terminal open for the remaining steps. The temporary path is printed; substitute your actual path/PID in another terminal when necessary.

```
umask 077
lab=$(mktemp -d /tmp/quinzee-internals.XXXXXX)
port=$(python3 - <<'PY'
import socket
with socket.socket() as sock:
    sock.bind(('127.0.0.1', 0))
    print(sock.getsockname()[1])
PY
)
openssl req -x509 -newkey rsa:2048 -nodes -days 1 \
  -keyout "$lab/server.key" -out "$lab/server.crt" \
  -subj '/CN=localhost' \
  -addext 'subjectAltName=DNS:localhost,IP:127.0.0.1' \
  -addext 'basicConstraints=critical,CA:TRUE' \
  >"$lab/certificate.log" 2>&1
bin/debug/quinzee.cloud --address 127.0.0.1 --port "$port" \
  --tls-cert "$lab/server.crt" --tls-key "$lab/server.key" \
  --ca-cert "$lab/server.crt" --data-dir "$lab/data" \
  --modules identity,resource_manager,storage \
  >"$lab/cloud.log" 2>&1 &
cloud_pid=$!
trap 'kill -TERM "$cloud_pid" 2>/dev/null || true; wait "$cloud_pid" 2>/dev/null || true' EXIT
printf 'Lab directory: %s\nCloud PID: %s\nHTTPS port: %s\n' "$lab" "$cloud_pid" "$port"
```

The temporary certificate is trusted only by commands explicitly given that file. It is a disposable test certificate, not a new machine-wide root. The port selection has a small bind/rebind race; if another process takes it, this cloud fails startup. Inspect this lab's log and retry with a new lab rather than killing the other listener. Only the selected modules and their required dependencies load; no AMQP, LDAP, DNS server or external runtime starts in this exercise.

Step 2: prove the process and endpoint agree

Allow a few seconds for startup, then run in the same terminal:

```
ps -p "$cloud_pid" -o pid,ppid,user,stat,comm
ps -L -p "$cloud_pid" -o tid,stat,wchan:28,comm
curl --fail --silent --show-error --cacert "$lab/server.crt" \
  "https://localhost:$port/health"
stat -c '%a %U:%G %n' "$lab" "$lab/data/credentials.json"
ls -l "/proc/$cloud_pid/fd"
```

Expect a running native process, several threads, a healthy HTTPS response and private bootstrap credentials. Do not print that credential file. You can see the listening descriptor, event-loop descriptors and module database files without reading their contents. Thread names may be identical; use TIDs and wait channels as clues, not as a stable module-to-thread API.

With iproute2 installed, `ss -ltnp` identifies the listening TCP endpoint when your permissions allow process attribution. A second terminal's plain curl without `--cacert` should not be used as the health test: it has not been given this lab's private trust root.

Step 3: observe file activity without exposing data

If inotify-tools is installed, watch only filenames and events in this lab:

```
inotifywait -m -r -e create,close_write,moved_to,delete "$lab/data"
```

In another terminal, point QHQ at this origin with this lab's `credentials.json` and `server.crt`, then create a synthetic resource group. Expect temporary-file writes and a move into the final state filename. Inotify reports events, not transaction contents or a durability proof. Stop the watcher with Ctrl+C when finished. A read-only health request should not be expected to create an ARM resource or continuously rewrite its state.

Step 4: sample system calls deliberately

If strace is available and local ptrace policy permits attaching to your own lab process, use a narrow syscall selection:

```
strace -f -p "$cloud_pid" \
  -e trace=epoll_wait,epoll_pwait,poll,ppoll,futex,fsync,fdatasync,rename,renameat,renameat2 \
  -o "$lab/scheduling-and-flush.trace"
```

Generate one health request or synthetic QHQ operation from another terminal, then Ctrl+C the tracer to detach. Event-loop waits, worker futex waits and flush/rename activity explain different parts of the request. Exact syscalls vary with the installed libraries and build. No

trace event is proof that every request must follow the same path.

This command omits read/write/send/receive payloads. Broad syscall tracing can capture passwords, tokens, settings or directory bind credentials after TLS has been decrypted in the process. Keep even these narrower traces private. Do not weaken ptrace policy, disable ASLR or add container privileges to make this exercise work; use the `/proc`, socket and application diagnostics when attaching is unavailable. Tracing also changes timing, so it is not a benchmark.

Step 5: follow the UI's Unix path

In the ordinary F5/Rasta environment, QHQ renders into the shared framebuffer file and Rasta displays it. Input returns over UDP. The UI process then schedules HTTPS work; it does not read provider JSON to make changes. The QHQ job timer polls completed work every 40 ms while jobs exist and delivers widget updates on the UI thread. A frozen viewer, a waiting HTTPS job and a stalled cloud worker are separate failure layers.

Inspect the viewer process, framebuffer file and UDP socket before concluding that a blank window means the cloud is down. Do not copy the live framebuffer while showing a secret: its pixels are ordinary data in a file. The offscreen screenshot backend uses the same real client/API path without requiring a viewer.

Step 6: observe graceful exit and retain evidence

Return to the original lab terminal:

```
kill -TERM "$cloud_pid"
wait "$cloud_pid"
trap - EXIT
printf 'Private lab evidence remains in %s\n' "$lab"
```

The wait should return successfully. The process and its listening socket disappear; private on-disk state remains. The trap is cleared because this instance has already been reaped. The lab directory contains credentials and key material: retain it privately only as long as needed, then deliberately remove that specific disposable directory. Do not use a wildcard over other instances' state. A clean shutdown test establishes this process lifecycle; it does not establish recovery after power loss or prove an external engine's backup is consistent.

Related reading

- The Book of Quinzee: Observe your application
- The Book of Quinzee: Troubleshoot a connection or workflow
- Quinzee Administrator’s Guide: Observe the platform and the application separately
- Quinzee Administrator’s Guide: Diagnose failures from the outside inward

- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: Practising Azure Administration
- Scenario overview: AI-Assisted Cloud Debugging

PUBLICATION NOTES

About this edition

This book is generated from the Quinzee Markdown manuals. Screenshots show the real application and QHQ in an isolated local environment; names, identifiers and ports vary between runs.

The PDF includes DejaVu font subsets. Their copyright and permission notice follows.

Fonts are (c) Bitstream (see below). DejaVu changes are in public domain. Glyphs imported from Arev fonts are (c) Tavmjong Bah (see below)

Bitstream Vera Fonts Copyright ----------

Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

Arev Fonts Copyright ----------

Copyright (c) 2006 by Tavmjong Bah. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the modifications to the Bitstream Vera Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Tavmjong Bah" or the word "Arev".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Tavmjong Bah Arev" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL TAVMJONG BAH BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the name of Tavmjong Bah shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from Tavmjong Bah. For further information, contact: tavmjong @ free . fr.

TeX Gyre DJV Math ---------- Fonts are (c) Bitstream (see below). DejaVu changes are in public domain.

Math extensions done by B. Jackowski, P. Strzelczyk and P. Pianowski (on behalf of TeX users groups) are in public domain.

Letters imported from Euler Fraktur from AMSfonts are (c) American Mathematical Society (see below). Bitstream Vera Fonts Copyright Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license (“Fonts”) and associated documentation files (the “Font Software”), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words “Bitstream” or the word “Vera”.

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the “Bitstream Vera” names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE. Except as contained in this notice, the names of GNOME, the GNOME Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the GNOME Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

AMSFonts (v. 2.2) copyright

The PostScript Type 1 implementation of the AMSFonts produced by and previously distributed by Blue Sky Research and Y&Y, Inc. are now freely available for general use. This has been accomplished through the cooperation of a consortium of scientific publishers with Blue Sky Research and Y&Y. Members of this consortium include:

Elsevier Science IBM Corporation Society for Industrial and Applied Mathematics (SIAM) Springer-Verlag American Mathematical Society (AMS)

In order to assure the authenticity of these fonts, copyright will be held by the American Mathematical Society. This is not meant to restrict in any way the legitimate use of the fonts, such as (but not limited to) electronic distribution of documents containing these fonts, inclusion of these fonts into other public domain or commercial font collections or computer applications, use of the outline data to create derivative fonts and/or faces, etc. However, the AMS does require that the AMS copyright notice be removed from any derivative versions of the fonts which have been altered in any way. In addition, to ensure the fidelity of TeX documents using Computer Modern fonts, Professor Donald Knuth, creator of the Computer Modern faces, has requested that any alterations which yield different font metrics be given a different name.

Id