Secure Tunnel System Reference

Generated from codebase

January 11, 2026

Contents

Part I

System Overview

Chapter 1

Scope and Objectives

1.1 Purpose and Non-goals

This book is a code-grounded reference for the drone–GCS post-quantum secure tunnel. It targets engineers who need to audit, extend, or operate the system without relying on tribal knowledge. In scope: network/data-plane behavior, handshake and AEAD framing, control-plane state, schedulers, telemetry, and power monitoring. Out of scope: airframe dynamics, flight controller internals, or generic PQC background beyond what the code uses.

1.2 Architecture at a Glance

The system is a split proxy: the GCS hosts the TCP handshake server and a UDP encrypted socket; the drone hosts the TCP handshake client and its own UDP encrypted socket. Plaintext adapters on each side front local apps/flight controllers. After the handshake, AEAD senders/receivers wrap UDP payloads with replay protection and epoch-based rekey triggers. Optional components include a TCP control server for rekey orchestration, schedulers that drive automated rekeys and telemetry, and power monitors for hardware characterization.

1.3 Threat Model and Assumptions

The design assumes an active network adversary who can intercept, replay, or inject packets on the untrusted link. TCP handshake authenticity is enforced via GCS signature and drone PSK; confidentiality and integrity are provided by the negotiated AEAD. Replay protection is required on both directions. Hosts and keys are assumed uncompromised; OS-level hardening is outside scope. NAT traversal is optional; strict peer matching is enabled by default.

Chapter 2

Configuration Surface

2.1 Global Defaults (core/config.py, CONFIG)

All runtime keys live in a single CONFIG map that defines network ports, host defaults, crypto knobs, scheduler parameters, and automation settings. Required keys and types are declared in `_REQUIRED_KEYS`; anything missing or mistyped fails validation. Defaults target LAN addresses (e.g., DRONE_HOST/GCS_HOST), enforce replay window bounds, set WIRE_VERSION=1, and leave DRONE_PSK empty to force env injection outside development.

2.2 Validation Logic (validate_config)

The validator checks presence and types for all required keys, bounds TCP/UDP ports to 1–65535, freezes WIRE_VERSION to 1, and requires REPLAY_WINDOW within [64, 8192]. Hosts must parse as IP literals; plaintext hosts may be restricted to loopback unless ALLOW_NON_LOOPBACK_PLAINTEXT is set. ENCRYPTED_DSCP is bounded to 0–63. DRONE_PSK must be hex and 32 bytes when provided, and is mandatory when ENV != "dev" (see core/config.py#L401-L470).

2.3 Environment Overrides ($_{a}pply\_env\_overrides$)

Environment variables listed in `_ENV_OVERRIDABLE` are parsed with type-aware casting (int/str/-bool/float) and merged before validation. Unsupported keys or malformed literals raise ConfigError, preventing partial override application (see core/config.py#L508-L549). This preserves a single source of truth while allowing per-host tuning without editing code.

2.4 Operational Profiles

The ENV guard treats "dev" as permissive (DRONE_PSK may be empty) and non-dev as strict (DRONE_PSK required). CONTROL_COORDINATOR_ROLE defaults to "gcs" but can be flipped to "drone" via env. ENABLE_TCP_CONTROL is opt-in; STRICT_UDP_PEER_MATCH and STRICT_HANDSHAKE_IP default to True to pin traffic to the authenticated peer unless NAT overrides are necessary.

2.5 Port Map

Port/Key	Role	Notes
TCP_HANDSHAKE_PORT	Handshake	46000 (GCS listens, drone connects).
UDP_DRONE_RX	Encrypted data plane (drone bind)	46012 (GCS sends here).
UDP_GCS_RX	Encrypted data plane (gcs bind)	46011 (drone sends here).
DRONE_PLAINTEXT_TX/RX	Local app interface (drone)	47003/47004 on 127.0.0.1; feeds FC/ proxy.
GCS_PLAINTEXT_TX/RX	Local app interface (gcs)	47001/47002 on 127.0.0.1; feeds ground
GCS_TELEMETRY_PORT	Telemetry (GCS → Drone)	52080 UDP feedback channel.
DRONE_CONTROL_PORT	Optional TCP control	48080; drone side when ABLE_TCP_CONTROL.
GCS_CONTROL_PORT	Optional TCP control	48080; GCS side when ABLE_TCP_CONTROL.
DRONE_TO_GCS_CTL_PORT	Encrypted control return	48181; drone-originated control over en plane.

Part II

Cryptographic Pipeline

Chapter 3

Suite Registry and Selection

3.1 Registry Structure (core/suites.py)

KEM, signature, and AEAD registries are explicit dictionaries with display names, OQS names, parameter IDs, and tokens. Alias maps normalize legacy names and shorthand; _build_alias_map folds each entry's aliases, oqs_name, display_name, and token into a case-insensitive lookup (core/suites.py#L430-L463). This guarantees consistent resolution even when OQS exposes variant capitalizations.

3.2 Suite Composition (build_suite_id, _compose_suite)

Suites are composed only from primitives sharing the same NIST level. `build_suite_id` resolves aliases and returns the canonical identifier `cs-<kem>-<aead>-<sig>` (core/suites.py#L467-L489). `_compose_suite` embeds KEM/SIG parameter IDs and the AEAD token, raising if levels mismatch to prevent downgrade by mix-level pairing (core/suites.py#L508-L545). Level-consistent matrices drive deterministic suite generation across all AEAD options.

3.3 Runtime Filtering (available_aead_tokens, enabled_sigs)

AEAD availability is probed at runtime: ChaCha20-Poly1305 is optional; Ascon-128a is gated by CONFIG flags and native/pyascon backends. `available_aead_tokens` returns only working tokens while exposing missing-reason details for diagnostics (core/suites.py#L390-L429). Signature availability is detected via OQS loader fallbacks, and `_prune_suites_for_runtime` removes suites whose signatures are not present, logging the removed set while keeping registry immutability (core/suites.py#L640-L712).

3.4 Header ID Mapping (header_ids_for_suite)

Each suite embeds deterministic header bytes (kem_id, kem_param_id, sig_id, sig_param_id). `header_ids_for_suite` extracts the tuple used by the AEAD header encoder; `header_ids_from_names` resolves IDs from negotiated names for async proxy fast-paths (core/suites.py#L756-L790). These IDs bind the encrypted stream to the authenticated handshake transcript.

3.5 Tables

Add a matrix of suite IDs with NIST level, AEAD token, and header IDs, generated from SUITES to avoid drift.

Chapter 4

Handshake Protocol

4.1 Key Roles

The GCS runs the TCP handshake server, signs the transcript, and decapsulates KEM ciphertexts; the drone runs the TCP client, verifies the signature, encapsulates, and authenticates with a PSK tag. Both sides capture timings for keygen/encap/decap/sign/verify to inform suite selection.

4.2 ServerHello Construction (build_server_hello)

The server selects a suite, generates a KEM keypair, and signs a transcript binding WIRE_VERSION, session ID, kem/sig names, and KEM public key (core/handshake.py#L115-L208). It seeds metrics with role/suite metadata and records wall/perf timings for KEM keygen and signature generation. The wire format is length-prefixed and includes version, name lengths, session ID, challenge, KEM pub, and signature.

4.3 ServerHello Verification (parse_and_verify_server_hello)

The client parses the length-prefixed message, reconstructs the transcript verbatim, and verifies the signature using the pinned GCS public key (core/handshake.py#L199-L290). Downgrade is prevented by embedding version and algorithm names into the signed transcript. Metrics capture signature verify timings and KEM public key sizes when provided.

4.4 PSK Auth Tag (_drone_psk_bytes)

After receiving the ciphertext, the GCS validates an HMAC-SHA256 tag over the raw ServerHello using the drone PSK (core/handshake.py#L305-L338, core/handshake.py#L440-L475). The PSK must be 32 bytes of hex and is mandatory outside "dev" environment; missing or malformed secrets abort the handshake.

4.5 Client Encapsulation (client_encapsulate)

The drone instantiates the negotiated KEM, encapsulates with the server's public key, and records ciphertext length plus timing for encap (core/handshake.py#L314-L343). Failures raise HandshakeError and free OQS resources when possible.

4.6 Server Decapsulation (server_decapsulate)

The GCS uses the ephemeral KEM object retained from ServerHello construction to decapsulate and logs timing plus ciphertext/shared-secret sizes (core/handshake.py#L345-L381). Ephemeral objects are freed and nulled to avoid reuse.

4.7 Key Derivation (derive_transport_keys)

Both sides derive 64 bytes via HKDF-SHA256 with salt "pq-drone-gcs—hkdf—v1" and info binding session ID, KEM name, and SIG name (core/handshake.py#L382-L428). The tuple is split into direction-specific keys; ordering differs by role to align send/receive perspectives. Metrics optionally record derivation timings.

4.8 Full Flows (server_gcs_handshake, client_drone_handshake)

`server_gcs_handshake` sends the ServerHello, receives ciphertext and PSK tag, verifies tag, decapsulates, derives keys, and finalizes metrics including total handshake time (core/handshake.py#L429-L525). `client_drone_handshake` enforces socket timeouts, parses and verifies ServerHello, checks suite match, generates ciphertext, computes HMAC tag, and sends both with length prefixes (core/handshake.py#L539-L653). Both paths are strict: any verification failure aborts with HandshakeVerifyError.

4.9 Message Formats

Document the length-prefixed ServerHello fields and the tuple (ciphertext length, ciphertext, HMAC tag) sent by the drone; include diagrams for clarity.

Chapter 5

AEAD Framing and Replay Protection

5.1 Header Format and Epochs

Headers pack version, kem/sig IDs, session ID, sequence number, and epoch using `HEADER_STRUCT = !BBBBB8sQB` (core/aead.py#L33-L45). The IV is not transmitted; only header+ciphertext are on the wire. Epoch increments accompany rekeys and forbid wrapping $255 \rightarrow 0$ without a new handshake.

5.2 Nonce Construction (_build_nonce)

Nonces are deterministic: one epoch byte plus 11-byte sequence, optionally right-padded when AEAD nonce length exceeds 12 (core/aead.py#L166-L191). This design removes IV-on-wire overhead while preserving unique nonces until REKEY_SEQ_THRESHOLD triggers.

5.3 Token Support (_canonicalize_aead_token, _instantiate_aead)

Tokens are validated against a supported set and reject retired aliases (core/aead.py#L47-L77). Instantiation enforces key sizes: AES-256-GCM and ChaCha20-Poly1305 require 32-byte keys; Ascon-128a accepts 16-byte keys with optional strict 16-byte enforcement (core/aead.py#L79-L165). Ascon prefers native C backend and falls back to pyascon with consistent error signaling.

5.4 Sender State Machine (Sender.encrypt, bump_epoch)

Sender validates wire version, IDs, session ID length, and epoch bounds at construction. Encryption packs the header, derives IV from epoch/seq, enforces REKEY_SEQ_THRESHOLD (default 2^{63}), and increments sequence after successful AEAD encryption (core/aead.py#L193-L286). `bump_epoch` resets sequence and forbids epoch wrap to prevent IV reuse.

5.5 Receiver State Machine (Receiver.decrypt, _check_replay)

Receiver mirrors validation and uses _check_replay to maintain a sliding bitmask window over the highest seen sequence, rejecting duplicates and too-old packets (core/aead.py#L288-L421). In strict mode it raises exceptions; in silent mode it returns None on header/session/epoch mismatch to avoid oracle leakage. Header binding ensures wrong session/IDs fail before decrypt.

5.6 Error Semantics

Header mismatches raise HeaderMismatch in strict mode; AeadAuthError wraps AEAD decrypt failures; ReplayError distinguishes duplicate vs. stale packets. SequenceOverflow signals proactive rekey needs when approaching IV exhaustion.

5.7 Tables

Add constants table covering REKEY_SEQ_THRESHOLD (default 2^{63}) and CONFIG.REPLAY_WINDOW (64–8192).

Chapter 6

Control Plane State Machine

6.1 Data Structures (ControlState, ControlResult)

ControlState holds role, coordinator_role, current_suite, thread lock, outbox queue, pending request map, FSM state, active request ID, last rekey metadata, last status, and stats counters; seen request IDs are capped via a deque to 256 (core/policy$_{e}ngine.py\#L20-L63).ControlResultaggregatesoutboundcontrolme$
$L74$).

6.2 Coordinator Role (set_coordinator_role, is_coordinator)

Coordinator role normalizes to “gcs” or “drone”; invalid inputs raise (core/policy$_{e}ngine.py\#L76-$
$L117).Helpersnormalizeconfigvaluesandcomparerolestodecidewhichsidedrivescommits, ensuringonlythecoord$

6.3 Prepare/Commit Flow (request_prepare, handle_control)

`request_prepare` moves the FSM to NEGOTIATING, enqueues a prepare_rekey with suite and rid, and bumps counters (core/policy$_{e}ngine.py\#L131-L157)$.`handle_control`$implementstwo-$
$phasecommitoverpacket{-}type0x02: thecoordinatorprocessesprepare\_ok/failandemitscommit\_rekey; thefollow$
$L259).Unknownormalformedmessagesareignoredwithnotes.$

6.4 Outbox Handling (enqueue_json)

Messages are queued via enqueue_json into a thread-safe queue consumed by the proxy loop, decoupling control decisions from I/O (core/policy$_{e}ngine.py\#L123-L129$).

6.5 Metrics and Stats

Counters track prepare sent/received and rekey outcomes; timestamps for last rekey suite/ms and last status are recorded for status introspection (core/policy$_{e}ngine.py\#L33-L63$).

Chapter 7

TCP Control Server (Optional)

7.1 Server Lifecycle (ControlTcpServer.start/stop)

The control server is a threaded TCP listener that binds with SO_REUSEADDR and 0.5s accept timeout; start() spins an accept loop thread and logs allowed peers, stop() sets a stop flag, closes the socket, and joins the thread (core/control$_{t}cp.py\#L22 - L105).Acceptandclientloopsaredefensive : errorsareloggedatdebuglevel, secretsareneverlogged.$

7.2 Authorization ($_{i}s\_allowed\_peer,_{i} s\_allowed\_rekey\_peer$)

Peers must be in an allow-list built from LAN/Tailscale hosts; loopback is always allowed for convenience (core/control$_{t}cp.py\#L157{-}L188).Rekeycommandsarestricter : onlydroneendpoints(andloopbackwhenser$
$L217$).

7.3 Command Set (ping, status, rekey)

Commands are newline-delimited JSON: ping/health returns ok and coordinator role; status snapshots ControlState fields; rekey validates peer authorization, coordinator role, and suite ID via get_suite, then calls request_prepare returning rid on success (core/control$_{t}cp.py\#L79{-}L154).Errorsareexplicit(u$

7.4 Factory Helpers (start_control_server_if_enabled)

start_control_server_if_enabled selects role-specific host/port keys, builds allow-lists, derives coordinator_role from config, constructs ControlTcpServer, and starts it if ENABLE_TCP_CONTROL is set (core/control$_{t}cp.py\#L219{-}L265).FailuretobindyieldsNone, leavingproxywithoutTCPcontrol.$

Chapter 8

Proxy Runtime (Selector-Based)

8.1 Process Layout

The proxy uses selectors (not asyncio) for deterministic I/O. Threads: optional TCP control server thread; optional manual console thread; main thread runs a selector loop over plaintext and encrypted UDP sockets with per-direction AEAD contexts. ControlPlane shares ControlState with lock-protected stats.

8.2 Handshake Invocation ($_{p}erform\_handshake$)

Role-gated: GCS binds TCP on TCP_HANDSHAKE_PORT, rate-limits via per-IP token bucket, enforces DRONE_HOST allowlist (STRICT_HANDSHAKE_IP), prunes rate-limit state, and performs server_gcs_handshake; drone connects outbound to GCS and runs client_drone_handshake (core/async$_{p}roxy.py\#L300-L470).I/OtimeoutsdefaulttoREKEY\_HANDSHAKE\_TIMEOUT(min10s).Retu$
$send/recvkeys, nonceseeds, sessionID, optionalkem/signames, peerUDPaddr, andhandshakemetrics.$

8.3 Socket Wiring ($_{s}etup\_sockets$)

Sockets are nonblocking UDP. Drone role binds encrypted at UDP_DRONE_RX and plaintext at DRONE_PLAINTEXT_TX (loopback by default); peers point to GCS$_{H}OST/UDP\_GCS\_RXandlocalplaintextRX$
$L560).ENCRYPTED\_DSCPisconvertedtoTOSwhenset.Plaintextin/outmayreusethesamesockettopreserveso$

8.4 AEAD Context Construction ($_{b}uild\_sender\_receiver$)

Describe how AeadIds are derived from suite or header IDs when available, selecting AEAD token from suite aead token or CONFIG.ENABLE_ASCON flags. (Fill with specific lines when added.)

8.5 Packet Flow (main loop)

Document selector loop: plaintext-¿encrypted path encrypts and forwards to peer; encrypted-¿plaintext path validates header, replay window, decrypts, and forwards to local app; control-plane messages (packet type 0x02 when ENABLE_PACKET_TYPE) are dispatched to policy_engine and may trigger rekey. (Cite run_proxy once sections are filled.)

8.6 Counters (ProxyCounters)

ProxyCounters tracks plaintext/encrypted in/out, drops (replay/auth/header/session/src), rekey outcomes, last rekey metadata, handshake metrics, and AEAD primitive timing histograms. It flattens OQS primitive metrics into part B fields (kem/sig times, sizes) for external reporting (core/async$_proxy.py\#L24 - L180$).

8.7 Manual Control ($_launch\_manual\_console$)

Interactive console (when enabled) to inspect counters and trigger actions; document once lines are located.

8.8 TCP Control Integration

If ENABLE_TCP_CONTROL, start_control_server_if_enabled creates per-role listener and bridges JSON commands into policy_engine state. Rekey worker watches ControlState, performs handshakes, and swaps AEAD contexts.

8.9 Error Handling

Drop reasons are classified before AEAD decrypt via $_parse\_header\_fields, distinguishingheader/ID/sessionmism$
$limitandunauthorizedpeersareloggedwithrole/expected/receivedcontext.$

Chapter 9

Process Lifecycle and Logging

9.1 ManagedProcess

TODO: Explain start/stop cross-platform safeguards.
Inline ref: core/process.py:ManagedProcess (lines TBD).

9.2 Registry Cleanup (kill_all_managed_processes)

TODO: Describe atexit behavior.
Inline ref: core/process.py:kill$_{a}ll_{m}anaged_{p}rocesses(linesTBD)$.

9.3 Logging Utilities

TODO: Summarize JsonFormatter, get_logger, configure_file_logger.
Inline ref: core/logging$_{u}tils.py(linesTBD)$.

9.4 Exception Taxonomy

TODO: List ConfigError, HandshakeError, AeadError, SequenceOverflow.
Inline ref: core/exceptions.py (lines TBD).

9.5 CLI Entry (core/run_proxy.py)

TODO: Sketch subcommands (init-identity, gcs, drone) and argument surfaces.
Inline ref: core/run$_{p}roxy.py(linesTBD)$.

Part III

Schedulers, Telemetry, and Policy

Chapter 10

Drone Scheduler (Controller)

10.1 Telemetry Intake (TelemetryListener)

TODO: Packet schema checks, logging, window feed.
Inline ref: sscheduler/sdrone.py:TelemetryListener (lines TBD).

10.2 Decision Context (DecisionContext)

TODO: Build DecisionInput and summarize telemetry.
Inline ref: sscheduler/sdrone.py:DecisionContext (lines TBD).

10.3 Policy Evaluation (TelemetryAwarePolicyV2)

TODO: Link to policy module and actions.
Inline ref: sscheduler/policy.py:TelemetryAwarePolicyV2 (lines TBD).

10.4 Action Execution (execute_action)

TODO: Switch/rekey workflows and cooldown handling.
Inline ref: sscheduler/sdrone.py:execute$_action(linesTBD)$.

10.5 Echo Path (UdpEchoServer)

TODO: Describe traffic loopback and counters.
Inline ref: sscheduler/sdrone.py:UdpEchoServer (lines TBD).

10.6 Proxy Management (DroneProxyManager)

TODO: Startup/shutdown, key lookup.
Inline ref: sscheduler/sdrone.py:DroneProxyManager (lines TBD).

10.7 Main Loop (run_scheduler)

TODO: Tick cadence, logging, failure handling.
Inline ref: sscheduler/sdrone.py:run$_scheduler(linesTBD)$.

10.8 Tables

TODO: Insert timing constants (cooldowns, TICK$_INTERVAL_S$).

Chapter 11

GCS Scheduler (Follower)

11.1 Control Server (ControlServer)

TODO: Command handling and telemetry loop.
Inline ref: sscheduler/sgcs.py:ControlServer (lines TBD).

11.2 Proxy Management (GcsProxyManager)

TODO: Start/stop logic and key material.
Inline ref: sscheduler/sgcs.py:GcsProxyManager (lines TBD).

11.3 Telemetry Sender

TODO: Sequenced UDP updates to drone.
Inline ref: sscheduler/sgcs.py:TelemetrySender (lines TBD).

11.4 MAVProxy Lifecycle (start_persistent_mavproxy)

TODO: Platform-specific flags and ports.
Inline ref: sscheduler/sgcs.py:start$_{p}ersistent_{m}avproxy(linesTBD)$.

11.5 Cleanup and Main

TODO: Environment cleanup strategy.
Inline ref: sscheduler/sgcs.py:cleanup$_{e}nvironment, main(linesTBD)$.

Chapter 12

Policy Engine (Telemetry-Aware)

12.1 Settings (load_settings)

TODO: Configurable thresholds and hysteresis.
Inline ref: sscheduler/policy.py:load$_{s}ettings(linesTBD)$.

12.2 Decision Input/Output (DecisionInput, PolicyOutput)

TODO: Fields and invariants.
Inline ref: sscheduler/policy.py:DecisionInput, PolicyOutput (lines TBD).

12.3 Scoring Logic (TelemetryAwarePolicyV2.evaluate)

TODO: Explain gates: telemetry validity, safety, link, rekey limits.
Inline ref: sscheduler/policy.py:TelemetryAwarePolicyV2.evaluate (lines TBD).

12.4 Suite Tiering (get_suite_tier)

TODO: Mapping of algorithms to tiers.
Inline ref: sscheduler/policy.py:get$_{s}uite_{t}ier(linesTBD)$.

12.5 Rekey Accounting (record_rekey)

TODO: Windowed counting for rekey rate limits.
Inline ref: sscheduler/policy.py:record$_{r}ekey(linesTBD)$.

Chapter 13

Telemetry Collection and Windows

13.1 GCS Metrics Collector (GcsMetricsCollector)

TODO: MAVLink/UDP capture, sliding windows, schema v1.
Inline ref: sscheduler/gcs$_{m}etrics.py : GcsMetricsCollector(linesTBD)$.

13.2 Local Monitor (LocalMonitor)

TODO: Thermal/battery/cpu ROC computation.
Inline ref: sscheduler/local$_{m}on.py : LocalMonitor(linesTBD)$.

13.3 Sliding Window (TelemetryWindow)

TODO: Confidence, jitter, missing sequence metrics.
Inline ref: sscheduler/telemetry$_{w}indow.py : TelemetryWindow(linesTBD)$.

13.4 Tables

TODO: Add message schema table for telemetry packets.

Part IV

Power and Monitoring

Chapter 14

Power Monitoring Backends

14.1 Interfaces (PowerMonitor, PowerSummary, PowerSample)

TODO: Define protocol expectations.
Inline ref: core/power$_{m}onitor.py(linesTBD)$.

14.2 INA219 Backend (Ina219PowerMonitor)

TODO: Sampling loop, CSV logging, sign detection.
Inline ref: core/power$_{m}onitor.py : Ina219PowerMonitor(linesTBD)$.

14.3 Raspberry Pi 5 hwmon (Rpi5PowerMonitor)

TODO: Channel discovery and scaling.
Inline ref: core/power$_{m}onitor.py : Rpi5PowerMonitor(linesTBD)$.

14.4 Pi PMIC (Rpi5PmicPowerMonitor)

TODO: vcgencmd parsing and power derivation.
Inline ref: core/power$_{m}onitor.py : Rpi5PmicPowerMonitor(linesTBD)$.

14.5 Factory (create_power_monitor)

TODO: Backend selection rules and fallbacks.
Inline ref: core/power$_{m}onitor.py : create_{p}ower_{m}onitor(linesTBD)$.

14.6 Tables

TODO: Backends vs dependencies vs sample rates.

Part V

Appendices

Chapter 15

Command-Line Interfaces

15.1 core/run_proxy.py

15.1.1 init-identity

TODO: Describe inputs and outputs.

15.1.2 gcs

TODO: Describe required keys and files.

15.1.3 drone

TODO: Describe required keys and files.

15.2 Scheduler Entrypoints

TODO: Summaries for sscheduler/sdrone.py and sscheduler/sgcs.py.

Chapter 16

Open Questions and Design Gaps

16.1 Design Intent Unclear

TODO: List ambiguous behaviors and propose clarifications.

16.2 Future Work

TODO: Outline next documentation tasks and test gaps.