Brand Protection Monitor (PoC) — Technical Scope & Orchestration Bible

Single definitive, implementation-ready technical specification.

Document date: 2026-02-01 | Version: 1.0 FINAL

Role and execution standard: The Architect — Master System Orchestrator v5.0.

Language: English only. Layout: flow-based. Margins: 36pt. Tables: wrapping, top-aligned, repeat headers on page breaks.

1. Introduction & Reference Framework

Purpose: This document is the authoritative technical bridge between product intent and engineering execution. It is designed to be consumed directly by implementation teams (backend, frontend, database, DevOps) and by automated agents without requiring any external clarification.

System summary (non-marketing, operational definition): A single-tenant PoC that periodically polls a single public Certificate Transparency (CT) log, extracts X.509 CN and SAN DNS names from recent certificate entries, performs case-insensitive substring matching against configured brand keywords, persists only matches (deduplicated), and exposes a UI for keyword management, triage/search of matches, dashboard monitoring state, and CSV export.

1.1 Source Inputs

Input Artifact	Role in this specification	Notes
PRD — Brand Protection Monitor (PoC) — Source of Truth	Defines product scope, modules, API contracts, state machines, NFRs.	Used as primary business/functional authority.
User Stories — Brand Protection Monitor (FULL)	Atomic, testable user stories and acceptance criteria.	Used to drive module mapping, validations, and traceability.
PostgreSQL Source of Truth — Full Database Design	Implementation-ready DDL, constraints, indexes, triggers, and DB traceability.	Used as database authority and performance baseline.
Rules_PDF_Generation.txt	Hard rules for PDF layout and content completeness.	Non-negotiable. This PDF is produced to comply.

1.2 Project Scope (In-Scope)

• Keyword CRUD (create, logical delete, list/search).

• CT monitoring engine with configurable scheduler interval and batch size; single-flight execution.

• CT range calculation using STH (get-sth) and fixed recent batch strategy.

• X.509 certificate parsing (CN + SAN DNS names), normalization, and substring matching.

• Deduplication of matches across cycles using unique key + UPSERT; update last_seen_at on every reappearance.

• Monitor status endpoint returning state + last run metrics.

• Matches endpoint with pagination, filtering, and sorting (server-side).

• CSV export endpoint with streaming output and filter-respecting behavior.

• Operational hardening: input sanitization, SQL parameterization, rate limiting, external HTTP timeouts.

• Observability: structured logs and persisted metrics per run.

1.3 Explicit Exclusions (Out-of-Scope)

• Authentication and authorization at runtime (no JWT, no login flows, no token refresh in PoC runtime paths).

• Multi-tenant support and per-tenant isolation.

• Multiple CT logs concurrently, incremental log cursoring beyond fixed recent batch strategy.

• Automated takedown workflows, notifications (email/webhook), or risk scoring beyond basic matching.

• UI/UX visual design deliverables beyond functional screen behaviors and state logic.

1.4 Technical Assumptions

• Single-tenant PoC: one operator persona; no runtime auth; system is still structured for later auth insertion points.

• One CT log endpoint configured in monitor_config; system fails hard if CT behavior changes/rotates.

• Unbounded fan-out per certificate is acceptable; controlled by batch_size and deduplication rather than caps.

• Database is the system source of truth for keywords, monitor state, runs, matches, and export events.

• No UI persistence across reloads; UI refetches keywords, status, and matches on load.

2. General Solution Vision

2.1 Core Concept

The PoC continuously produces an evidence dataset of potentially abusive domains that have obtained TLS certificates, using CT as the signal source. The system is intentionally optimized for operational triage: fast list/search/filter/export of matches, and explicit monitor health visibility.

2.2 Business Engine Logic (Global Rules)

• Matching rule: case-insensitive substring contains(keyword) applied to candidate domains derived from CN and SAN DNS names.

• Persistence rule: store only matches; do not store non-matching certificate entries.

• Deduplication rule: unique key = (cert_fingerprint_sha256, matched_keyword, matched_domain).

• Temporal rule: first_seen_at is immutable; last_seen_at updates on every reappearance (including within the same run).

• Keyword deletion rule (audit decision): deleted keywords are ignored by all filters and exports; retained only for audit in DB.

• Monitor execution rule: single-flight; a cycle cannot start while monitor_state.state == running.

• Failure rule: if a cycle fails after persisting some matches, do not roll back those matches; finalize run as error; rely on dedup + last_seen for correctness.

• Export rule: exports include committed matches as of export start, including those committed during an active run; export is non-blocking.

2.3 Implementation Constraints

• No runtime auth: endpoints are assumed internal/PoC environment; still implement rate limiting and input validation.

• No multi-log: ct_log_base_url is single value; future phases may add multi-log ingestion and per-log cursors.

• No global transaction per monitor cycle: commit matches per UPSERT or micro-batch to avoid holding long transactions.

3. System Architecture (Orchestration Level)

3.1 Logical Layering

Layer	Responsibilities	Key Artifacts / Interfaces
Scheduler	Triggers cycle every poll_interval_seconds; enforces single-flight via DB state transition.	monitor_state singleton; monitor_config.poll_interval_seconds; logs with run_id correlation.
CT Consumer	Calls CT endpoints (get-sth, get-entries) with timeouts; transforms responses into entry list.	HTTP client with connect/read timeouts; error codes mapped to cycle_fatal_error.
Parser	Extracts X.509 fields from certificates; collects CN and SAN DNS names; normalizes domains.	Parsing library; error classification; parse_error_count.
Matcher	Case-insensitive substring search across active keywords; produces match tuples.	keyword normalization; match_found event.
Persistence	UPSERT matches; append-only monitor_run; updates monitor_state; records exports.	PostgreSQL; unique indexes; triggers; optimistic locking where relevant.
REST API	Exposes /keywords, /matches, /export.csv, /monitor/status, /healthz, /readyz.	Go Gin handlers; validation middleware; error envelope.
React UI	Dashboard, keyword management, match triage table, export modal; error/empty/loading states.	React+TS, TanStack Router, Zustand, TanStack Query, Zod, Axios.

3.2 Data Flow Strategy (Request → Persistence)

• UI requests are stateless (no auth) and always read fresh data from API (no persistence across reload).

• API validates request payloads (server-side) and performs parameterized SQL via pgx/v5.

• CT cycle writes are append-only for runs and UPSERT-based for matches; state is updated in monitor_state.

• Exports stream rows directly from DB using a cursor/rows iterator; export metadata is recorded in exports table.

3.3 Identity & Session Management

PoC runtime explicitly excludes authentication/authorization. Therefore: no JWT issuance/verification, no refresh token rotation, and no auth middleware in request path. The database may include users/refresh_tokens tables for forward compatibility, but they are not referenced by runtime flows.

3.4 Observability Layer

• Structured JSON logs in backend, correlated by run_id for monitor cycles.

• Persisted run metrics in monitor_run for UI and debugging without dependence on external log aggregation.

• monitor_state contains user-facing error code/message fields for dashboard display.

• Health checks: /healthz (process alive) and /readyz (DB connectivity and monitor_state query).

4. Tech Stack & Engineering Justification

4.1 Backend Stack

• Go + Gin: high-performance HTTP server; middleware architecture; concurrency-friendly for periodic cycles.

• pgx/v5: native PostgreSQL driver with pooling and prepared statements; avoids ORM ambiguity; supports parameterized queries.

• go-playground/validator: request struct validation; complements DB constraints; enforces keyword length and formats.

• Zap: structured logging; low overhead; consistent log schemas for run correlation.

• HTTP client timeouts: connect/read timeouts for CT endpoints; prevents runaway cycles.

4.2 Frontend Stack

• React + TypeScript: typed UI, resilient state handling, contract enforcement.

• TanStack Router: typed routing; enables protected routes in future (even though auth is excluded now).

• Zustand: minimal global store (ephemeral in PoC); no persistence to localStorage/sessionStorage.

• TanStack Query: server state caching and refetch control for /monitor/status, /keywords, /matches.

• Zod: runtime validation for both user inputs and API responses; invalid responses downgrade UI to controlled error state.

• Axios: request management; no auth interceptors required in PoC.

• Tailwind CSS: utility styling; not part of scope beyond functional layout and state-driven rendering.

5. Module Orchestration (Master Section)

Each module below includes: technical purpose; user story mapping; backend logic flow (atomic steps); database interaction matrix; dependencies and contracts; Zod and Zustand integration notes (frontend).

5.1. KEYWORD MANAGEMENT

A. Technical Purpose

Provide operators with full lifecycle management of brand keywords used by the matcher. Keywords are the only operator-managed configuration driving match detection. Deleted keywords are retained only for audit but are excluded from all filters and exports (audit decision).

B. User Story Mapping

• HU-KW-01 -> impacts KEYWORD MANAGEMENT module behavior and contracts.

• HU-KW-02 -> impacts KEYWORD MANAGEMENT module behavior and contracts.

• HU-KW-03 -> impacts KEYWORD MANAGEMENT module behavior and contracts.

C. Detailed Backend Logic Flow (Step-by-step)

1. HTTP request enters Gin router.

2. Security middleware: rate limit check (global policy).

3. Parse JSON body for POST /keywords; extract value.

4. Server validation: reject missing value key; reject non-string types.

5. Server validation: trim; reject empty or whitespace-only.

6. Server validation: enforce max length 64; reject longer inputs.

7. Compute normalized_value = lower(trim(value)).

8. Start DB transaction (short-lived).

9. Attempt INSERT into keywords(value, normalized_value, status='active', is_deleted=false).

10. If unique constraint on normalized_value where is_deleted=false is violated: map to 409 Conflict (duplicate).

11. Commit transaction; return created keyword payload with keyword_id, value, normalized_value, status, created_at.

12. For DELETE /keywords/{keyword_id}: parse path param; validate numeric.

13. Start DB transaction.

14. Update keywords set status='inactive', is_deleted=true, deleted_at=NOW() where keyword_id=? and is_deleted=false.

15. If 0 rows affected: return 404 Not Found (already deleted or missing).

16. Commit transaction; return success envelope.

17. For GET /keywords: parse optional q, page, page_size; validate page_size in {10,25,50} or default 25.

18. Query keywords where is_deleted=false (only active dataset); apply substring search on normalized_value ILIKE %q% if provided.

19. Order by keyword_id DESC; return items and total for pagination.

Validation Rules (Server + DB)

• Reject empty or whitespace-only values (AC: HU-KW-01 AC1).

• Normalize: trim leading/trailing whitespace; compute normalized_value = lower(trim(value)) (AC: HU-KW-01 AC2).

• Reject duplicates case-insensitively by enforcing unique normalized_value where is_deleted=false (AC: HU-KW-01 AC3).

• Length: 1..64 characters (AC: HU-SEC-01 AC1); enforced by DB CHECK and server validation.

Frontend Contracts: Zod + Zustand

• Zod schema validates POST /keywords request body before submit: { value: string } with trim + min(1) + max(64).

• Zod schema validates GET /keywords response: items[] containing keyword_id, value, normalized_value, status, created_at, updated_at.

• Zustand store is ephemeral: no persistence; keyword list is held in-memory only until reload.

• On success: TanStack Query invalidates 'keywords' query; UI refreshes list immediately.

• On error: UI displays inline message and preserves user input (does not clear form).

D. Database Interaction Matrix

Category	Tables	Primary Filters / Keys	Notes
Read	keywords	Varies by endpoint; always is_deleted=false where applicable.	Read path optimized by indexes and pg_trgm where configured.
Write	keywords	Writes are atomic; no long-running transactions across full cycle.	Soft delete and optimistic locking semantics preserved by triggers.

E. Dependencies & Contracts

• Depends on unique index on normalized_value to prevent duplicates.

• Depends on soft delete semantics (is_deleted + deleted_at) to preserve audit trail.

5.2. CT MONITORING ENGINE

A. Technical Purpose

Periodically ingest recent CT entries from a single configured log, parse certificates, perform substring matching, and persist deduplicated matches with temporal evidence. This module is the core data-producing pipeline.

B. User Story Mapping

• HU-MON-01 -> impacts CT MONITORING ENGINE module behavior and contracts.

• HU-MON-02 -> impacts CT MONITORING ENGINE module behavior and contracts.

• HU-MON-03 -> impacts CT MONITORING ENGINE module behavior and contracts.

C. Detailed Backend Logic Flow (Step-by-step)

1. Scheduler wakes every poll_interval_seconds (monitor_config).

2. Start transaction T1 to acquire single-flight lock.

3. Read monitor_state singleton row (id=1) FOR UPDATE.

4. If monitor_state.state != 'idle': rollback T1 and exit (no overlap allowed).

5. If idle: update monitor_state.state='running', last_run_at=NOW(), clear last_error_* fields; commit T1.

6. Insert monitor_run row with state='running', started_at=NOW(). Retrieve run_id.

7. Fetch CT STH (get-sth) with ct_connect_timeout_ms and ct_read_timeout_ms.

8. If CT STH call fails: mark cycle fatal -> finalize run as error; update monitor_state to error; release lock by setting state='error'; exit.

9. Compute range_end = STH.tree_size (or tree_size-1 depending on CT indexing; in PoC define inclusive end).

10. Compute range_start = max(0, range_end - batch_size). Persist range_start/range_end to monitor_run.

11. Fetch CT entries for [range_start, range_end] with get-entries calls (may be chunked if API requires).

12. For each returned entry: attempt parse of certificate chain -> leaf certificate.

13. If parse fails: increment parse_error_count; continue (non-fatal).

14. Extract CN and SAN DNS names; ignore IP SANs or other SAN types.

15. Normalize candidate domains: lower; strip trailing dot; optional punycode normalization if library supports; ensure length <= 253.

16. Read active keyword set from DB (keywords where is_deleted=false and status='active'). Cache in-memory for the cycle.

17. For each domain and each keyword: if domain contains keyword (case-insensitive substring), emit match tuple.

18. Compute cert_fingerprint_sha256 for certificate (SHA-256 hex).

19. UPSERT match into matched_certificates using unique key (fingerprint, matched_keyword, matched_domain) where is_deleted=false.

20. UPSERT update path ALWAYS sets last_seen_at=NOW() and last_run_id=run_id (including within same run if encountered again).

21. Increment processed_count per certificate; increment match_count per emitted match tuple.

22. After processing batch: compute duration_ms, ct_latency_ms, db_latency_ms; update monitor_run.state='success', finished_at=NOW(), counters.

23. Finalize monitor_state: transaction T2 updates monitor_state.state='idle', last_success_at=NOW(), last_run_at preserved, clear last_error_* fields.

24. If any fatal DB error occurs mid-run after some UPSERTs: set monitor_run.state='error' with error_code/message; monitor_state.state='error' and last_error_at=NOW(). Persisted matches remain committed.

Database Interaction Matrix

Operation	Read Tables	Write Tables	Key Constraints/Indexes Used	Notes
Acquire single-flight lock	monitor_state	monitor_state	monitor_state_singleton; row lock	Atomic transition idle→running in DB.
Load runtime config	monitor_config	-	singleton id=1	Hot reloadable config without redeploy.
Create run record	-	monitor_run	idx_monitor_run_started_at_desc; state enum	Append-only semantics.
Persist match (UPSERT)	keywords (cached)	matched_certificates	ux_match_key_active; gin_matches_domain_trgm	last_seen_at always updated.

Frontend Contracts: Zod + Zustand

• No direct UI writes for this module; it produces data consumed via /monitor/status and /matches.

• UI polling strategy: TanStack Query refetch /monitor/status on interval (e.g., 10s) or on window focus.

• Zod validates /monitor/status response; if invalid, UI enters controlled error state and offers retry.

• Zustand stores only ephemeral flags (e.g., last known status) during session; cleared on reload.

D. Database Interaction Matrix

Category	Tables	Primary Filters / Keys	Notes
Read	monitor_config, monitor_state, keywords	Varies by endpoint; always is_deleted=false where applicable.	Read path optimized by indexes and pg_trgm where configured.
Write	monitor_state, monitor_run, matched_certificates	Writes are atomic; no long-running transactions across full cycle.	Soft delete and optimistic locking semantics preserved by triggers.

E. Dependencies & Contracts

• Depends on CT log availability and correctness; failures are fatal per cycle (fail-hard PoC strategy).

• Depends on keywords table as configuration input; uses only non-deleted, active keywords.

• Depends on monitor_state singleton row existence (id=1). System initialization must seed this row.

5.3. MATCHES & TRIAGE

A. Technical Purpose

Expose a deterministic, filterable, paginated list of deduplicated matches for operator investigation. This module is read-only; it must be fast and stable under typical query patterns (time-sort + substring filters).

B. User Story Mapping

• HU-MAT-01 -> impacts MATCHES & TRIAGE module behavior and contracts.

• HU-MAT-02 -> impacts MATCHES & TRIAGE module behavior and contracts.

C. Detailed Backend Logic Flow (Step-by-step)

1. HTTP GET /matches enters Gin router.

2. Rate limiting middleware enforces per-IP limits (especially for search/filter usage).

3. Parse query params: page, page_size, keyword, q, issuer, new_only, date_from, date_to, sort.

4. Validate page_size is one of allowed values (10/25/50) with sane default.

5. Validate sort is one of: first_seen_desc (default), last_seen_desc, domain_asc.

6. Build parameterized SQL (no string interpolation).

7. Base filter: matched_certificates.is_deleted=false.

8. If keyword present: filter matched_keyword = keyword (exact).

9. If q present: filter matched_domain ILIKE %q% OR issuer ILIKE %q% (substring).

10. If issuer present: filter issuer ILIKE %issuer%.

11. If date_from/date_to present: apply first_seen_at range filter (inclusive boundaries defined).

12. If new_only=true: filter first_seen_at >= (last successful run started_at or finished_at) OR use last_run_id to compute newness; PoC uses derivable rule based on first_seen_at in last cycle window.

13. Apply ORDER BY according to sort; apply LIMIT/OFFSET for pagination.

14. Return payload with items[] and total count for pagination.

15. UI highlights matched substring within domain field (presentation logic), without altering stored data.

Frontend Contracts: Zod + Zustand

• Zod validates /matches response shape: items[] with required fields (domain, matched_keyword, matched_field, issuer, not_before, not_after, first_seen_at, last_seen_at).

• TanStack Query key includes filter params; changing filters triggers refetch with debounce (300ms).

• Zustand stores current filter UI state only in-memory (ephemeral); reset on reload.

• Invalid API response or network error: UI shows persistent banner with retry; does not crash the view.

D. Database Interaction Matrix

Category	Tables	Primary Filters / Keys	Notes
Read	matched_certificates	Varies by endpoint; always is_deleted=false where applicable.	Read path optimized by indexes and pg_trgm where configured.
Write	(none)	Writes are atomic; no long-running transactions across full cycle.	Soft delete and optimistic locking semantics preserved by triggers.

E. Dependencies & Contracts

• Depends on database indexes to meet dashboard and triage latency expectations.

5.4. EXPORT & REPORTING

A. Technical Purpose

Generate RFC4180-compliant CSV exports of matches based on active filters, using streaming output to avoid high memory usage. Record export events for operational traceability.

B. User Story Mapping

• HU-EXP-01 -> impacts EXPORT & REPORTING module behavior and contracts.

C. Detailed Backend Logic Flow (Step-by-step)

1. HTTP GET /export.csv enters Gin router.

2. Rate limiting middleware applies stricter limits to prevent abuse (per-IP).

3. Parse query params identical to /matches (filters + sort).

4. Validate params; build parameterized SQL selecting same dataset as /matches.

5. Start export event record in exports: requested_at=NOW(), filters=JSONB snapshot of applied filters.

6. Open DB rows cursor; stream CSV header row first (RFC4180).

7. For each DB row: write CSV line with proper quoting/escaping; do not buffer entire dataset in memory.

8. At export completion: update exports.completed_at=NOW(), rows_exported=count, duration_ms.

9. If export fails mid-stream: still update exports with completed_at and partial rows if possible; return appropriate HTTP error only if headers not already sent.

10. Consistency guarantee: export includes all committed rows as of export start; it is non-blocking and may include committed matches written by an active run.

Frontend Contracts: Zod + Zustand

• Export is initiated from a modal that summarizes currently applied filters (chips).

• No client-side CSV assembly; browser downloads directly from /export.csv?{params}.

• UI disables export button while download is preparing to avoid duplicates.

• Errors show controlled message; user can retry.

D. Database Interaction Matrix

Category	Tables	Primary Filters / Keys	Notes
Read	matched_certificates	Varies by endpoint; always is_deleted=false where applicable.	Read path optimized by indexes and pg_trgm where configured.
Write	exports	Writes are atomic; no long-running transactions across full cycle.	Soft delete and optimistic locking semantics preserved by triggers.

E. Dependencies & Contracts

• Depends on matched_certificates query stability and indexes to support streaming without long locks.

• Depends on RFC4180 escaping rules in CSV writer.

5.5. DASHBOARD & MONITOR STATE

A. Technical Purpose

Expose monitor health and latest run metrics. Provide explicit states (idle/running/error), surfaced error fields, and stable UI behaviors for loading/error/empty states.

B. User Story Mapping

• HU-DASH-01 -> impacts DASHBOARD & MONITOR STATE module behavior and contracts.

C. Detailed Backend Logic Flow (Step-by-step)

1. HTTP GET /monitor/status enters Gin router.

2. Read monitor_state singleton (id=1) and latest monitor_run (ORDER BY started_at DESC LIMIT 1) for metrics_last_run.

3. Compose response: state (idle/running/error), last_run_at, last_success_at, last_error_code, last_error_message, metrics_last_run object.

4. HTTP GET /healthz returns process alive (no DB dependency).

5. HTTP GET /readyz validates DB connectivity by querying monitor_state (id=1) with timeout; failure returns non-200.

6. UI on load: fetch /monitor/status and initial /matches with default sort first_seen_desc and page_size=25.

7. UI state logic: loading skeleton; persistent error banner if status.state=error or if /matches fails; empty state CTAs.

Frontend Contracts: Zod + Zustand

• Zod validates /monitor/status shape; invalid payload triggers controlled error state.

• TanStack Query polls /monitor/status periodically; pauses when tab is hidden if desired.

• Zustand stores only ephemeral UI flags (dismissed banners, current view state) during session; reset on reload.

D. Database Interaction Matrix

Category	Tables	Primary Filters / Keys	Notes
Read	monitor_state, monitor_run	Varies by endpoint; always is_deleted=false where applicable.	Read path optimized by indexes and pg_trgm where configured.
Write	monitor_state, monitor_run	Writes are atomic; no long-running transactions across full cycle.	Soft delete and optimistic locking semantics preserved by triggers.

E. Dependencies & Contracts

• Depends on database indexes to meet dashboard and triage latency expectations.

5.6. SECURITY & HARDENING

A. Technical Purpose

Provide baseline PoC hardening: input validation and sanitization, SQL parameterization, rate limiting, and CT HTTP timeouts.

B. User Story Mapping

• HU-SEC-01 -> impacts SECURITY & HARDENING module behavior and contracts.

• HU-SEC-02 -> impacts SECURITY & HARDENING module behavior and contracts.

C. Detailed Backend Logic Flow (Step-by-step)

1. All endpoints pass through request validation middleware (type checks, bounds, required fields).

2. Keyword input validation: length 1..64, trim normalization, duplicate prevention via DB unique constraint.

3. Output sanitization: UI escapes all text content to prevent XSS; highlight logic must not inject HTML unsafely.

4. SQL injection prevention: all DB access uses parameterized queries (pgx), never string concatenation.

5. Rate limiting: basic per-IP token bucket; stricter for /export.csv and high-cardinality search endpoints.

6. CT HTTP timeouts: connect and read timeouts applied; failures are cycle-fatal; monitor_state transitions to error.

7. Error handling: standardized error envelope for JSON endpoints; CSV endpoint uses HTTP status if headers not sent.

D. Database Interaction Matrix

Category	Tables	Primary Filters / Keys	Notes
Read	(none)	Varies by endpoint; always is_deleted=false where applicable.	Read path optimized by indexes and pg_trgm where configured.
Write	(none)	Writes are atomic; no long-running transactions across full cycle.	Soft delete and optimistic locking semantics preserved by triggers.

E. Dependencies & Contracts

• Depends on database indexes to meet dashboard and triage latency expectations.

6. Database Specification (PostgreSQL)

The database schema below is implementation-ready. All table and index definitions are designed to support PoC behavior: soft deletion, deduplicated match persistence, monitor run history, export tracing, and operational querying performance.

6.1 Extensions and Enums

```
-- Extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pgcrypto";
CREATE EXTENSION IF NOT EXISTS "pg_trgm";

-- Enums
CREATE TYPE keyword_status AS ENUM ('active', 'inactive');
CREATE TYPE monitor_state_enum AS ENUM ('idle', 'running', 'error');
CREATE TYPE monitor_run_state_enum AS ENUM ('running', 'success', 'error');
CREATE TYPE matched_field_enum AS ENUM ('cn', 'san', 'both');
```

6.2 Shared Trigger Functions

```
CREATE OR REPLACE FUNCTION trigger_set_updated_at()
RETURNS TRIGGER AS $$
BEGIN
NEW.updated_at = NOW();
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION trigger_bump_version_number()
RETURNS TRIGGER AS $$
BEGIN
NEW.version_number = OLD.version_number + 1;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

6.3 Tables, Constraints, Indexes, and Triggers

Note: even if PoC runtime excludes authentication, the schema includes users for audit attribution and future hardening. PoC runtime code MUST NOT depend on users table for request authorization.

Table: users

```
CREATE TABLE IF NOT EXISTS users (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
email VARCHAR(255) UNIQUE NOT NULL,
password_hash VARCHAR(255) NOT NULL,
full_name VARCHAR(255) NOT NULL,
is_active BOOLEAN DEFAULT TRUE NOT NULL,
created_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
last_login_at TIMESTAMPTZ,
deleted_at TIMESTAMPTZ,
deleted_by UUID REFERENCES users(id) ON DELETE SET NULL,
is_deleted BOOLEAN DEFAULT FALSE NOT NULL,
version_number INTEGER DEFAULT 1 NOT NULL,
CONSTRAINT email_format CHECK (
email ~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
)
);
CREATE INDEX IF NOT EXISTS idx_users_email ON users(email) WHERE is_deleted = FALSE;
CREATE INDEX IF NOT EXISTS idx_users_active ON users(is_active) WHERE is_active = TRUE AND
is_deleted = FALSE;
CREATE INDEX IF NOT EXISTS idx_users_deleted ON users(is_deleted);
CREATE TRIGGER set_updated_at_users BEFORE UPDATE ON users FOR EACH ROW EXECUTE FUNCTION
trigger_set_updated_at();
CREATE TRIGGER bump_version_users BEFORE UPDATE ON users FOR EACH ROW EXECUTE FUNCTION
trigger_bump_version_number();
```

Table: keywords

```
CREATE TABLE IF NOT EXISTS keywords (
keyword_id BIGSERIAL PRIMARY KEY,
value TEXT NOT NULL,
normalized_value TEXT NOT NULL,
status keyword_status NOT NULL DEFAULT 'active',
created_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
deleted_at TIMESTAMPTZ,
created_by UUID REFERENCES users(id) ON DELETE SET NULL,
updated_by UUID REFERENCES users(id) ON DELETE SET NULL,
deleted_by UUID REFERENCES users(id) ON DELETE SET NULL,
is_deleted BOOLEAN DEFAULT FALSE NOT NULL,
version_number INTEGER DEFAULT 1 NOT NULL,
CONSTRAINT kw_value_length CHECK (char_length(value) BETWEEN 1 AND 64),
CONSTRAINT kw_normalized_length CHECK (char_length(normalized_value) BETWEEN 1 AND 64),
CONSTRAINT kw_status_deleted_consistency CHECK (
(is_deleted = FALSE AND deleted_at IS NULL) OR
(is_deleted = TRUE AND deleted_at IS NOT NULL)
)
);
CREATE UNIQUE INDEX IF NOT EXISTS ux_keywords_normalized_active ON keywords(normalized_value) WHERE
is_deleted
= FALSE;
CREATE INDEX IF NOT EXISTS idx_keywords_status ON keywords(status) WHERE is_deleted = FALSE;
CREATE TRIGGER set_updated_at_keywords BEFORE UPDATE ON keywords FOR EACH ROW EXECUTE FUNCTION
trigger_set_updated_at();
CREATE TRIGGER bump_version_keywords BEFORE UPDATE ON keywords FOR EACH ROW EXECUTE FUNCTION
trigger_bump_version_number();
```

Table: monitor_config

```
CREATE TABLE IF NOT EXISTS monitor_config (
id SMALLINT PRIMARY KEY DEFAULT 1,
ct_log_base_url TEXT NOT NULL,
poll_interval_seconds INT NOT NULL DEFAULT 60,
batch_size INT NOT NULL DEFAULT 100,
ct_connect_timeout_ms INT NOT NULL DEFAULT 2000,
ct_read_timeout_ms INT NOT NULL DEFAULT 5000,
created_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
created_by UUID REFERENCES users(id) ON DELETE SET NULL,
updated_by UUID REFERENCES users(id) ON DELETE SET NULL,
deleted_at TIMESTAMPTZ,
deleted_by UUID REFERENCES users(id) ON DELETE SET NULL,
is_deleted BOOLEAN DEFAULT FALSE NOT NULL,
version_number INTEGER DEFAULT 1 NOT NULL,
CONSTRAINT monitor_config_singleton CHECK (id = 1),
CONSTRAINT poll_interval_positive CHECK (poll_interval_seconds BETWEEN 5 AND 86400),
CONSTRAINT batch_size_positive CHECK (batch_size BETWEEN 1 AND 10000),
CONSTRAINT timeout_positive CHECK (ct_connect_timeout_ms > 0 AND ct_read_timeout_ms > 0),
CONSTRAINT mc_deleted_consistency CHECK (
(is_deleted = FALSE AND deleted_at IS NULL) OR
(is_deleted = TRUE AND deleted_at IS NOT NULL)
)
);
CREATE TRIGGER set_updated_at_monitor_config BEFORE UPDATE ON monitor_config FOR EACH ROW EXECUTE
FUNCTION
trigger_set_updated_at();
CREATE TRIGGER bump_version_monitor_config BEFORE UPDATE ON monitor_config FOR EACH ROW EXECUTE
FUNCTION
trigger_bump_version_number();
```

Table: monitor_state

```
CREATE TABLE IF NOT EXISTS monitor_state (
id SMALLINT PRIMARY KEY DEFAULT 1,
state monitor_state_enum NOT NULL DEFAULT 'idle',
last_run_at TIMESTAMPTZ,
last_success_at TIMESTAMPTZ,
last_error_at TIMESTAMPTZ,
last_error_code TEXT,
last_error_message TEXT,
created_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
created_by UUID REFERENCES users(id) ON DELETE SET NULL,
updated_by UUID REFERENCES users(id) ON DELETE SET NULL,
deleted_at TIMESTAMPTZ,
deleted_by UUID REFERENCES users(id) ON DELETE SET NULL,
is_deleted BOOLEAN DEFAULT FALSE NOT NULL,
version_number INTEGER DEFAULT 1 NOT NULL,
CONSTRAINT monitor_state_singleton CHECK (id = 1),
CONSTRAINT ms_deleted_consistency CHECK (
(is_deleted = FALSE AND deleted_at IS NULL) OR
(is_deleted = TRUE AND deleted_at IS NOT NULL)
)
);
CREATE TRIGGER set_updated_at_monitor_state BEFORE UPDATE ON monitor_state FOR EACH ROW EXECUTE
FUNCTION
trigger_set_updated_at();
CREATE TRIGGER bump_version_monitor_state BEFORE UPDATE ON monitor_state FOR EACH ROW EXECUTE
FUNCTION
trigger_bump_version_number();
```

Table: monitor_run

```
CREATE TABLE IF NOT EXISTS monitor_run (
run_id BIGSERIAL PRIMARY KEY,
started_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
finished_at TIMESTAMPTZ,
ct_tree_size BIGINT,
range_start BIGINT,
range_end BIGINT,
processed_count INT NOT NULL DEFAULT 0,
match_count INT NOT NULL DEFAULT 0,
parse_error_count INT NOT NULL DEFAULT 0,
duration_ms INT,
ct_latency_ms INT,
db_latency_ms INT,
state monitor_run_state_enum NOT NULL DEFAULT 'running',
error_code TEXT,
error_message TEXT,
created_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
created_by UUID REFERENCES users(id) ON DELETE SET NULL,
updated_by UUID REFERENCES users(id) ON DELETE SET NULL,
deleted_at TIMESTAMPTZ,
deleted_by UUID REFERENCES users(id) ON DELETE SET NULL,
is_deleted BOOLEAN DEFAULT FALSE NOT NULL,
version_number INTEGER DEFAULT 1 NOT NULL,
CONSTRAINT counts_non_negative CHECK (processed_count >= 0 AND match_count >= 0 AND
parse_error_count >= 0),
CONSTRAINT timing_non_negative CHECK (
(duration_ms IS NULL OR duration_ms >= 0) AND
(ct_latency_ms IS NULL OR ct_latency_ms >= 0) AND
(db_latency_ms IS NULL OR db_latency_ms >= 0)
),
CONSTRAINT finished_after_started CHECK (finished_at IS NULL OR finished_at >= started_at),
CONSTRAINT mr_deleted_consistency CHECK (
(is_deleted = FALSE AND deleted_at IS NULL) OR
(is_deleted = TRUE AND deleted_at IS NOT NULL)
)
);
CREATE INDEX IF NOT EXISTS idx_monitor_run_started_at_desc ON monitor_run(started_at DESC) WHERE
is_deleted =
```

```
FALSE;
CREATE INDEX IF NOT EXISTS idx_monitor_run_state ON monitor_run(state) WHERE is_deleted = FALSE;
CREATE TRIGGER set_updated_at_monitor_run BEFORE UPDATE ON monitor_run FOR EACH ROW EXECUTE
FUNCTION
trigger_set_updated_at();
CREATE TRIGGER bump_version_monitor_run BEFORE UPDATE ON monitor_run FOR EACH ROW EXECUTE FUNCTION
trigger_bump_version_number();
```

Table: matched_certificates

```
CREATE TABLE IF NOT EXISTS matched_certificates (
match_id BIGSERIAL PRIMARY KEY,
cert_fingerprint_sha256 TEXT NOT NULL,
matched_domain TEXT NOT NULL,
matched_keyword TEXT NOT NULL,
matched_field matched_field_enum NOT NULL,
issuer TEXT,
not_before TIMESTAMPTZ,
not_after TIMESTAMPTZ,
source_ct_log TEXT NOT NULL,
first_seen_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
last_seen_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
last_run_id BIGINT REFERENCES monitor_run(run_id) ON DELETE SET NULL,
created_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
created_by UUID REFERENCES users(id) ON DELETE SET NULL,
updated_by UUID REFERENCES users(id) ON DELETE SET NULL,
deleted_at TIMESTAMPTZ,
deleted_by UUID REFERENCES users(id) ON DELETE SET NULL,
is_deleted BOOLEAN DEFAULT FALSE NOT NULL,
version_number INTEGER DEFAULT 1 NOT NULL,
CONSTRAINT fingerprint_non_empty CHECK (char_length(cert_fingerprint_sha256) BETWEEN 16 AND 128),
CONSTRAINT domain_non_empty CHECK (char_length(matched_domain) BETWEEN 1 AND 253),
CONSTRAINT keyword_non_empty CHECK (char_length(matched_keyword) BETWEEN 1 AND 64),
CONSTRAINT validity_window CHECK (not_before IS NULL OR not_after IS NULL OR not_after >=
not_before),
CONSTRAINT last_seen_after_first CHECK (last_seen_at >= first_seen_at),
CONSTRAINT mc_deleted_consistency CHECK (
(is_deleted = FALSE AND deleted_at IS NULL) OR
(is_deleted = TRUE AND deleted_at IS NOT NULL)
)
);
CREATE UNIQUE INDEX IF NOT EXISTS ux_match_key_active ON
matched_certificates(cert_fingerprint_sha256,
matched_keyword, matched_domain) WHERE is_deleted = FALSE;
CREATE INDEX IF NOT EXISTS idx_matches_first_seen_desc ON matched_certificates(first_seen_at DESC)
WHERE
is_deleted = FALSE;
CREATE INDEX IF NOT EXISTS idx_matches_last_seen_desc ON matched_certificates(last_seen_at DESC)
WHERE
is_deleted = FALSE;
CREATE INDEX IF NOT EXISTS idx_matches_keyword ON matched_certificates(matched_keyword) WHERE
is_deleted =
FALSE;
CREATE INDEX IF NOT EXISTS gin_matches_domain_trgm ON matched_certificates USING GIN
(matched_domain
gin_trgm_ops) WHERE is_deleted = FALSE;
CREATE INDEX IF NOT EXISTS gin_matches_issuer_trgm ON matched_certificates USING GIN (issuer
gin_trgm_ops)
WHERE is_deleted = FALSE AND issuer IS NOT NULL;
CREATE TRIGGER set_updated_at_matched_certificates BEFORE UPDATE ON matched_certificates FOR EACH
ROW EXECUTE
FUNCTION trigger_set_updated_at();
CREATE TRIGGER bump_version_matched_certificates BEFORE UPDATE ON matched_certificates FOR EACH ROW
EXECUTE
FUNCTION trigger_bump_version_number();
```

Table: exports

```
CREATE TABLE IF NOT EXISTS exports (
export_id BIGSERIAL PRIMARY KEY,
requested_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
completed_at TIMESTAMPTZ,
filters JSONB NOT NULL DEFAULT '{}'::jsonb,
rows_exported INT,
duration_ms INT,
created_by UUID REFERENCES users(id) ON DELETE SET NULL,
created_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_at TIMESTAMPTZ DEFAULT NOW() NOT NULL,
updated_by UUID REFERENCES users(id) ON DELETE SET NULL,
deleted_at TIMESTAMPTZ,
deleted_by UUID REFERENCES users(id) ON DELETE SET NULL,
is_deleted BOOLEAN DEFAULT FALSE NOT NULL,
version_number INTEGER DEFAULT 1 NOT NULL,
CONSTRAINT export_finished_after_started CHECK (completed_at IS NULL OR completed_at >=
requested_at),
CONSTRAINT rows_non_negative CHECK (rows_exported IS NULL OR rows_exported >= 0),
CONSTRAINT export_timing_non_negative CHECK (duration_ms IS NULL OR duration_ms >= 0),
CONSTRAINT exports_deleted_consistency CHECK (
(is_deleted = FALSE AND deleted_at IS NULL) OR
(is_deleted = TRUE AND deleted_at IS NOT NULL)
)
);
CREATE INDEX IF NOT EXISTS gin_exports_filters ON exports USING GIN (filters jsonb_path_ops) WHERE
is_deleted
= FALSE;
CREATE INDEX IF NOT EXISTS idx_exports_requested_at_desc ON exports(requested_at DESC) WHERE
is_deleted =
FALSE;
CREATE TRIGGER set_updated_at_exports BEFORE UPDATE ON exports FOR EACH ROW EXECUTE FUNCTION
trigger_set_updated_at();
CREATE TRIGGER bump_version_exports BEFORE UPDATE ON exports FOR EACH ROW EXECUTE FUNCTION
trigger_bump_version_number();
```

6.4 Reference UPSERT Pattern for Matches (Idempotent)

```
INSERT INTO matched_certificates (
cert_fingerprint_sha256, matched_domain, matched_keyword, matched_field,
issuer, not_before, not_after, source_ct_log,
first_seen_at, last_seen_at, last_run_id, created_by, updated_by
) VALUES (
$1, $2, $3, $4::matched_field_enum, $5, $6, $7, $8,
NOW(), NOW(), $9, $10, $10
)
ON CONFLICT (cert_fingerprint_sha256, matched_keyword, matched_domain)
WHERE is_deleted = FALSE
DO UPDATE SET
last_seen_at = NOW(),
last_run_id = EXCLUDED.last_run_id,
updated_by = EXCLUDED.updated_by
RETURNING match_id, first_seen_at, last_seen_at;
```

7. REST API Specification (Contracts + Validations)

All JSON endpoints return application/json. CSV export endpoint returns text/csv with streaming semantics. Authentication is excluded from runtime paths; therefore, no Authorization header is required in PoC.

GET /monitor/status

Purpose: Dashboard monitor health and last run metrics.

Query/Params: (none)

Field	Type	Description
state	string	idle | running | error
last_run_at	timestamp|null	Last cycle start timestamp.
last_success_at	timestamp|null	Last successful cycle.
last_error_code	string|null	Error code if state=error.
last_error_message	string|null	Short error message if state=error.
metrics_last_run	object|null	processed_count, match_count, parse_error_count, duration_ms, ct_latency_ms, db_latency_ms

HTTP	Error Code	When
500	DB_UNAVAILABLE	Database query failure.

GET /keywords

Purpose: List/search active keywords (non-deleted).

Query/Params: q?: substring search; page?: int; page_size?: 10|25|50

Field	Type	Description
items	array	Keyword rows
total	int	Total count for pagination

HTTP	Error Code	When
400	INVALID_QUERY	Invalid pagination params.
500	DB_ERROR	Database query failure.

POST /keywords

Purpose: Create a new keyword with normalization and dedupe.

Query/Params: (none)

Field	Type	Description
keyword_id	int	Created keyword id
value	string	Original value as provided
normalized_value	string	lower(trim(value))
status	string	active
created_at	timestamp	Creation time

HTTP	Error Code	When
400	VALIDATION_ERROR	Empty/too long/invalid type.
409	DUPLICATE_KEYWORD	normalized_value already exists (active, non-deleted).
500	DB_ERROR	Database failure.

DELETE /keywords/{keyword_id}

Purpose: Soft delete keyword (audit retained) and exclude from all filters and exports.

Query/Params: (none)

Field	Type	Description
ok	boolean	true

HTTP	Error Code	When
400	INVALID_PATH_PARAM	keyword_id not numeric.
404	NOT_FOUND	keyword does not exist or already deleted.
500	DB_ERROR	Database failure.

GET /matches

Purpose: Paginated triage list of deduplicated matches.

Query/Params: page, page_size, keyword, q, issuer, date_from, date_to, new_only, sort

Field	Type	Description
items	array	Match rows with required fields
total	int	Total count for pagination

HTTP	Error Code	When
400	INVALID_QUERY	Invalid filters/sort/pagination.
500	DB_ERROR	Database failure.

GET /export.csv

Purpose: Stream CSV export of matches respecting current filters.

Query/Params: Same as /matches

Field	Type	Description
(stream)	text/csv	RFC4180 rows; deterministic ordering.

HTTP	Error Code	When
400	INVALID_QUERY	Invalid filters.
429	RATE_LIMITED	Too many exports per IP.
500	EXPORT_ERROR	Export failed before headers sent.

GET /healthz

Purpose: Liveness probe.

Query/Params: (none)

Field	Type	Description
ok	boolean	true

GET /readyz

Purpose: Readiness probe (DB connectivity + monitor_state query).

Query/Params: (none)

Field	Type	Description
ok	boolean	true

HTTP	Error Code	When
503	NOT_READY	DB unreachable or query timeout.

8. Business Rules, State Machines, and Automations

8.1 Monitor State Machine (monitor_state.state)

State	Meaning	Entry Conditions	Exit Conditions	Visible UI Behavior
idle	No active CT cycle executing.	Default after boot; after successful cycle finalization.	scheduler acquires lock and transitions to running.	Dashboard shows 'Idle'; allows triage and exports.
running	A CT cycle is actively executing.	Atomic transaction updates state idle→running; monitor_run created.	Cycle ends success -> idle; fatal error -> error.	Dashboard shows 'Running'; metrics may show last completed run; exports still allowed.
error	Monitor cannot proceed without manual intervention or next tick recovery.	Cycle fatal error (CT/DB) sets state=error, last_error_code/message, last_error_at.	Next tick attempts a new cycle; on success set idle and clear error fields.	Dashboard shows error banner with code/message and retry guidance; triage and keyword CRUD should still work if DB is healthy.

8.2 Monitor Run Lifecycle (monitor_run.state)

State	Meaning	When Set	Required Fields	Notes
running	Run started, not finalized.	Immediately after insert.	run_id, started_at	May persist partial metrics during execution.
success	Run completed without fatal errors.	After processing all entries.	finished_at, counts, duration_ms, ct_latency_ms, db_latency_ms	parse_error_count may be >0 (non-fatal).
error	Run terminated due to fatal error (CT/DB).	On fatal error detection.	error_code, error_message, finished_at optional	Persisted matches remain committed (no global rollback).

8.3 Keyword Lifecycle (keywords.status + soft delete)

Attribute	Allowed Values	Rules	Downstream Effects
status	active | inactive	active means eligible for matching; inactive means excluded. In PoC, deletion sets inactive.	Inactive/deleted keywords are not used by matcher.
is_deleted	boolean	Soft delete flag; when true must have deleted_at not null (DB consistency CHECK).	Deleted keywords are ignored by all filters and exports; retained for audit only.
normalized_value	lower(trim(value))	Unique where is_deleted=false; prevents duplicates case-insensitively.	Ensures operator cannot create duplicates via casing.

8.4 Automations / Triggers (System-level)

AUTO-ID	Trigger	Condition	Action	Observability
AUTO-01	scheduler_tick	Every poll_interval_seconds	Attempt single-flight; start CT cycle.	monitor_run.started_at; logs correlated by run_id.
AUTO-02	ct_fetch_complete	Valid CT get-entries response	Process entries; parse and match.	processed_count; ct_latency_ms.
AUTO-03	match_found	CN/SAN contains keyword	UPSERT match; update last_seen_at; set first_seen_at if new.	match_count; 'new' derivable from first_seen_at vs last cycle.
AUTO-04	cycle_end	After processing batch	Finalize monitor_run metrics; update monitor_state.	monitor_state.last_run_at; monitor_run.duration_ms.

9. Domain Stress Tests & Edge-Case Analysis

This section enumerates failure and edge scenarios that must be handled deterministically by the PoC. Each item includes expected behavior and persistence impact.

Scenario	Expected Runtime Behavior	State/Metric Updates	Persistence Rules
CT get-sth timeout	CT consumer hits connect/read timeout; mark cycle fatal.	monitor_run: state=error; error_code=CT_TIMEOUT; monitor_state.state=error; last_error_at set.	No global rollback; previously committed matches remain.
CT returns malformed entry	Parser fails for specific entry.	Increment parse_error_count; continue run; run may still succeed.	No match persistence for that entry.
Duplicate domains within a certificate	Same SAN DNS appears multiple times.	UPSERT executed multiple times; last_seen_at updates each time; no duplicate rows.	Unique key prevents duplicates.
Keyword deleted mid-run (race)	Operator deletes keyword while cycle cached keyword list.	PoC behavior: cycle uses cached active keywords at load time for that run; deletion affects next run.	Persisted matches from that run remain; exported/filtered views ignore deleted keywords going forward.
DB error mid-run after partial UPSERTs	DB connection drop or constraint failure during persistence.	Finalize run as error; monitor_state=error; keep already committed matches.	Next cycles dedup and update last_seen.
Export during active run	Operator triggers /export.csv while monitor_state=running.	Export streams committed rows snapshot at export start; may include committed rows from active run.	Export event recorded in exports table with filters snapshot.
Unbounded fan-out	A single cert matches many keywords and many SANs.	Persist one row per (fingerprint, keyword, domain) without cap; controlled by batch_size and dedupe.	Potential high match_count; ensure indexes and streaming behavior prevent memory blowups.
Keyword contains non-ASCII	Operator inputs Unicode (e.g., accented characters).	Validation allows within length bounds; normalization lower/trim; matching uses case-insensitive substring semantics.	Consider punycode normalization for domains if implemented; otherwise match on raw lowercase.

10. Decision Record (Key Decisions and Rationale)

Decision	Rationale
No runtime authentication	PoC is single-tenant and internal; prioritize pipeline correctness and operability. Security handled via validation, parameterized SQL, and rate limits.
Single CT log; fail-hard on evolution	Reduces complexity for PoC; makes failures visible and forces manual reconfiguration instead of silently degrading signal quality.
Single-flight via DB-backed state row	Prevents overlapping runs across restarts or multiple instances; DB is source of truth; row lock provides atomicity.
No global transaction per cycle	Avoid long transactions; allow partial persistence; rely on idempotent UPSERT and last_seen updates for eventual correctness.
Dedup key includes domain	Prevents collapsing multiple abusive domains in a single cert; preserves operability per domain.
Match persistence only (no raw CT storage)	Keeps storage small and focuses on actionable evidence; avoids storing massive irrelevant CT entries.
Exports stream with RFC4180	Prevents memory blowups and enables large exports; ensures compatibility with spreadsheet tooling.
No UI state persistence	Backend is source of truth; reduces complexity and avoids stale states; aligns with PoC simplicity.

11. Operations: Runbook-Level Considerations

11.1 Deployment/Environment Assumptions (PoC)

• Runs locally (docker-compose) or PoC environment with direct DB access.

• CT log base URL configured in monitor_config.ct_log_base_url.

• Scheduler interval and batch size configured in monitor_config.

• Monitoring: dashboard uses /monitor/status; SRE can use /readyz for DB readiness.

11.2 Failure Modes and Recovery

• CT failures: monitor_state transitions to error; next scheduler tick retries; success clears error fields and returns to idle.

• DB failures: /readyz fails; UI shows banner; keyword CRUD and match reads fail deterministically with DB_ERROR codes.

• Partial match persistence: accepted; operator sees some matches even if run errored; subsequent runs update last_seen and add new matches.

11.3 Observability Fields (Minimum Log Schema)

Field	Type	Meaning
run_id	int	Correlates all events within a cycle.
event	string	e.g., scheduler_tick, ct_fetch, parse_error, match_upsert, cycle_end, cycle_fatal_error.
duration_ms	int	Overall or per-step timing.
processed_count	int	Certificates processed.
match_count	int	Match tuples generated/UPSERTed.
parse_error_count	int	Non-fatal parse failures.
error_code	string|null	Only on fatal errors.

12. Performance, Quality, and Risk Analysis

12.1 Performance Drivers and Index Strategy

• Time-sorted triage queries: B-Tree on first_seen_at DESC and last_seen_at DESC.

• Exact filters: B-Tree on matched_keyword.

• Substring search: pg_trgm GIN indexes on matched_domain and issuer.

• Export filter snapshots: JSONB GIN index on exports.filters.

• Run history: B-Tree on monitor_run.started_at DESC and state.

12.2 Risks and Mitigations

Risk	Impact	Mitigation (PoC)	Mitigation (Post-PoC)
CT endpoint changes/rotates	Pipeline stops (fail-hard).	Surface error state; manual reconfiguration.	Multi-log support; per-log cursor; automated health checks.
Unbounded fan-out creates large match_count	High write volume; larger exports.	Control via batch_size; UPSERT idempotency; index tuning.	Partition matched_certificates; async processing; parallelization.
Long exports affect DB latency	Potential read contention.	Streaming + indexes; rate limiting.	Read replica; export jobs; background task queue.
No auth could be abused	Unauthorized access in non-isolated env.	Assume internal PoC; enforce rate limit; restrict network.	Add auth middleware and JWT; RBAC; audit events.

13. Evolution, Scalability, and Future Extensions

13.1 Roadmap-Compatible Extensions (Explicit)

• Multi-log ingestion: support multiple CT logs concurrently; persist log identity per match (already stored in source_ct_log).

• Incremental cursoring beyond fixed recent batch: persist last processed tree size per log; handle gaps.

• Alerting: email/webhook notifications on new matches; dedup per keyword/domain/time window.

• Risk scoring: heuristics on issuer reputation, domain similarity (typosquatting), cert validity windows.

• Auth: add JWT/SSO and multi-tenant isolation; protected routes in router; audit_events for compliance-grade changes.

13.2 Scaling Recommendations (DB and App)

• Partitioning: add monthly RANGE partitions on monitor_run.started_at and matched_certificates.first_seen_at when counts reach millions.

• Read replicas: route dashboards and exports to replicas to isolate write jitter.

• Materialized views: daily aggregates by keyword/issuer for reporting at scale.

• Batch parallelism: parse entries concurrently with bounded worker pool; preserve ordering only where needed for determinism.

14. Traceability

14.1 User Story → API → DB Traceability Matrix

User Story	Primary Endpoints	DB Tables	Key Constraints/Indexes	Acceptance Criteria Coverage Notes
HU-KW-01 Create keyword	POST /keywords	keywords	ux_keywords_normalized_active; kw_value_length	No empty; normalized; no duplicates; persisted; UI refresh via query invalidation.
HU-KW-02 Delete keyword	DELETE /keywords/{id}	keywords	soft delete consistency CHECK	Soft delete with deleted_at; historical matches remain; deleted keywords ignored by filters/exports (audit decision).
HU-KW-03 List/search keywords	GET /keywords	keywords	idx_keywords_status; substring ILIKE	Pagination; substring search; visible status.
HU-MON-01 Periodic cycle	(internal scheduler)	monitor_config, monitor_state, monitor_run	idx_monitor_run_started_at_desc	Configurable scheduler; run metrics persisted; CT errors handled.
HU-MON-02 CT range	(internal CT consumer)	monitor_run	range_start/range_end fields	Uses STH; start/end computed and persisted.
HU-MON-03 Dedup matches	(internal matcher)	matched_certificates	ux_match_key_active; UPSERT	Unique key; UPSERT; updates last_seen_at always.
HU-MAT-01 List matches	GET /matches	matched_certificates	idx_matches_first_seen_desc	Fields complete; highlight in UI; server-side pagination.
HU-MAT-02 Filter/sort matches	GET /matches	matched_certificates	GIN trigram; time indexes	Multiple filters; configurable sort; new_only filter.
HU-EXP-01 Export CSV	GET /export.csv	matched_certificates, exports	ordering indexes; gin_exports_filters	Respects filters; streaming; RFC4180 compliance.
HU-DASH-01 Monitor status	GET /monitor/status	monitor_state, monitor_run	singleton + latest run query	idle/running/error; visible errors; metrics visible.
HU-SEC-01 Sanitization	(all endpoints)	(all)	parameterized SQL	Validation; UI escaping; SQL parameterization.
HU-SEC-02 Rate limit/timeouts	(all endpoints)	(none)	(rate limiter config)	Rate limit by IP; CT timeouts enforced.

15. Final Acceptance Checklist (Completeness Verification)

• Single definitive PDF: no drafts, no placeholders, no external dependency for understanding.

• English only throughout the document.

• Flow-based layout only; no absolute positioning; automatic page breaks used by Platypus flowables.

• Margins fixed at 36pt on all sides; no content outside printable area.

• All tables use wrapped Paragraph cells, VALIGN=TOP; no truncation; repeat header row across pages.

• All mandatory content items present: scope/inputs, resolved assumptions, stress tests, decision records, explicit models, flows, complete specs, validations, traceability, operations, performance/risk, evolution strategy, final checklist.

• Agent role sections fully executed: architecture, tech stack, module orchestration with Zod/Zustand notes in each flow, DB matrices, state transitions, concurrency, rollback strategy, and scaling.