1 WiSACS V1 Prototype Specification

2 Document Status

This Pandoc Markdown specification records the local implementation contract for the WiSACS V1 prototype. It is repository-maintained and is intended to be edited with the implementation changes it describes.

The prototype exposes a deterministic local API under:

`/api/v1`

The implementation is simulation-first by default. It models discovery, situational awareness, RF/IP network inventory, and advisory interoperability recommendations without real radio control, real active RF sensing, certified emergency-services deployment, or automatic live network bridging.

Artifact	Role
`docs/spec/*.md`	Repository-maintained Pandoc Markdown specification parts.
`backend/WISACS_API.md`	Code-adjacent implementation traceability map.
`backend/tests/WISACS_API_COVERAGE.md`	Test-adjacent verification traceability map.
`scripts/build_wisacs_spec_docs.py`	Cross-platform Python/Pandoc renderer for normal PDF, review PDF, LaTeX, and HTML outputs.

3 Update Workflow

1. Update the relevant `docs/spec/*.md` source when the normative WiSACS contract changes.
2. Update backend/WISACS_API.md when implementation files, symbols, or production gaps move.
3. Update backend/tests/WISACS_API_COVERAGE.md when verification files, test names, or coverage expectations change.
4. Render review output with line numbers when needed:

```
uv run python scripts/build_wisacs_spec_docs.py --mode review --format pdf
```

4 Pandoc Markdown Conventions

This specification uses Pandoc Markdown rather than plain CommonMark so the source can carry stable requirement anchors, document metadata, pipe tables, definition lists, fenced code blocks, and PDF-oriented review controls without requiring authors to write LaTeX.

Requirement headings use explicit identifiers such as:

```
### WISACS-API-001: Health And Readiness {#WISACS-API-001}
```

Code-adjacent documents link back to those stable identifiers and use relative links to source files, tests, and generated artifacts.

5 Normative Language

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in RFC 2119 and RFC 8174 when, and only when, they appear in uppercase.

This document defines two conformance profiles:

Simulation Profile The deterministic local API implemented in this repository. A Simulation Profile implementation MAY use seeded demo users, local SQLite state, simulated incident data, and caller-supplied observations, but it MUST expose those limitations.

Production Profile A target operational WiSACS implementation. A Production Profile implementation MUST satisfy Simulation Profile requirements that are not explicitly marked simulation-only, and MUST additionally satisfy production identity, persistence, cryptographic, operational-safety, audit-retention, and incident-data governance requirements.

Conforming implementations MUST NOT silently mix profiles. The repository implementation currently claims Simulation Profile behavior only.

6 Specification Parts

The complete rendered specification is assembled by build_wisacs_spec_docs.py in this order:

1. System definition and workflow.
2. API contract.
3. Data model.
4. Security posture.
5. SCA alignment.
6. Render deployment.
7. Traceability workflow.
8. Implementation traceability.
9. Verification coverage.

That split keeps reviewer-facing specification text in `docs/spec/` while placing implementation-facing documentation next to the code it describes.

6.1 Purpose

WiSACS defines a prototype system for disaster-response wireless situation awareness and coordination. The system helps responders and incident coordinators answer three early-scene questions:

1. Who is present?
2. Where are personnel, devices, and observed RF/IP systems?
3. Which communication paths appear technically plausible enough for human review?

This specification describes V1 prototype behavior. It is not a production emergency-services certification, a live network-bridging standard, or a formal SCA conformance profile.

6.2 Scope

V1 covers discovery, inventory, visualization support, advisory interoperability recommendations, and reviewable SCA-ready metadata. V1 intentionally excludes automatic radio control, automatic cross-network bridging, active RF scanning hardware control, production identity federation, and operational policy automation.

6.3 Terms and Definitions

Incident An emergency-scene operating context that bounds responders, devices, networks, observations, recommendations, and event history.

Responder A person, crew role, or operational unit participating in an incident.

Field device A radio, handset, tablet, gateway, router, SDR sensor, laptop, or other deployed system that may communicate, observe RF conditions, or provide services.

Network An observed RF, IP, voice, mesh, satellite, or other communication system present at the incident.

Network attachment Evidence that a field device is associated with a network, including optional address, signal, reachability, and observation metadata.

RF observation A report of spectrum activity at a frequency with optional bandwidth, RSSI, protocol guess, modulation guess, confidence, source, and location.

Service endpoint A TCP/IP-reachable service reported for a device or network.

Interoperability recommendation An advisory result that summarizes a candidate communication path and its supporting evidence for human review.

SCA-ready descriptor A draft JSON representation of device/component metadata that is shaped to support later SCA descriptor work but is

not a conformance artifact.

6.4 Informative System Overview

The V1 prototype uses a cloud-hosted API as the shared incident state. Field users, simulation tools, or later hardware probes submit observations. The backend stores these observations, computes advisory recommendations, and exposes inventory data to client applications. The single-origin public deployment serves documentation at the root path, a lightweight HTML/SVG browser operational dashboard at `/visualization`, and the operational API under `/api/v1`. The Godot visualization can run as an installed client against the same hosted API, while the Flutter scaffold remains available for later cross-platform client development.

The system is deliberately conservative. WiSACS may identify that multiple devices share an IP network, that a gateway is attached to more than one network, or that a voice-only path must be preserved. WiSACS does not activate a bridge or change live network policy in V1.

6.5 Normative Requirements

6.5.1 WISACS-SYS-001: Incident-Scoped Inventory

The system MUST maintain an incident-scoped inventory of responders, field devices, networks, RF observations, and reachable service endpoints.

Implementation references (`WISACS-SYS-001`): models.py, routes.py, demo.py, wisacs_probe.py, pages.py, WiSACSOldMontrealResponse.gd, main.dart.

Verification references (`WISACS-SYS-001`): test_api.py, test_wisacs_probe.py.

6.5.2 WISACS-SYS-002: Location Data For Client Display

The system MUST expose enough location data for clients to show responders, devices, observations, and incident operating areas on a map or fallback grid.

Implementation references (`WISACS-SYS-002`): models.py, demo.py, pages.py, WiSACSOldMontrealResponse.gd, main.dart.

Verification references (`WISACS-SYS-002`): test_api.py.

6.5.3 WISACS-SYS-003: Advisory Interoperability Only

The V1 system MUST NOT automatically bridge live networks; it MAY recommend candidate interoperability actions for human review.

Implementation references (`WISACS-SYS-003`): interop.py, routes.py.

Verification references (`WISACS-SYS-003`): test_api.py.

6.6 Non-Normative Notes

V1 discovery is manual entry plus simulator/probe ingestion. Active RF scanning, production identity federation, and live data relay are deferred.

6.7 Informative V1 Workflow

A typical demonstration flow is:

1. An incident coordinator authenticates to the API.
2. The demo scenario or a probe tool posts responders, field devices, networks, attachments, RF observations, and service endpoints.
3. The backend computes interoperability recommendations from the current incident inventory.
4. The client displays the operating area, responders, devices, networks, observations, and recommendations.
5. Reviewers inspect each recommendation and its evidence before any real-world action is taken outside WiSACS.

7 WiSACS API Specification

7.1 Purpose

The WiSACS API is the system boundary between clients, probe tools, stored incident state, and interoperability recommendation logic. The API is designed to be inspectable through OpenAPI and usable by both the Flutter client and lightweight command-line tools.

7.2 API Conventions

All operational REST endpoints are under `/api/v1`. Root-level health and readiness endpoints are provided for hosting platforms that expect simple unauthenticated checks. JSON is the default request and response format. Identifier fields are opaque strings and should not be parsed by clients.

Bearer-token authentication is used for V1 operational endpoints. Demo users are seeded by the prototype for local review, but production deployments should replace this with an incident-appropriate identity and authorization design.

7.3 Roles

Admin A user with prototype administrative access.

Commander A user allowed to perform privileged setup actions such as loading the demo scenario.

Responder A field user expected to contribute or inspect incident observations.

Observer A read-oriented review user.

7.4 Resource Families

The API is grouped around the incident lifecycle:

- Health and readiness checks support deployment and monitoring.
- Authentication endpoints issue and inspect bearer tokens.
- Inventory endpoints maintain incidents, responders, devices, networks, attachments, RF observations, and service endpoints.
- Recommendation endpoints compute advisory communication options from the current incident state.
- Import endpoints seed repeatable review scenarios.
- SCA endpoints expose draft metadata for later standards alignment.
- WebSocket endpoints publish incident update notifications.

7.5 Normative Requirements

7.5.1 WISACS-API-001: Health And Readiness

The API MUST expose health and readiness endpoints for local checks and hosted deployment checks.

Implementation references (`WISACS-API-001`): main.py, routes.py.

Verification references (`WISACS-API-001`): test_api.py.

7.5.2 WISACS-API-002: Authentication Identity

The API MUST provide a login endpoint and an authenticated current-user endpoint.

Implementation references (`WISACS-API-002`): routes.py, deps.py, security.py.

Verification references (`WISACS-API-002`): test_api.py.

7.5.3 WISACS-API-003: Incident Endpoints

The API MUST expose incident creation and listing endpoints.

Implementation references (`WISACS-API-003`): routes.py, schemas.py.

Verification references (`WISACS-API-003`): test_api.py.

7.5.4 WISACS-API-004: Core Inventory Endpoints

The API MUST expose responder, field-device, and network inventory endpoints.

Implementation references (`WISACS-API-004`): routes.py, schemas.py.

Verification references (`WISACS-API-004`): test_api.py.

7.5.5 WISACS-API-005: Observation And Endpoint Ingestion

The API MUST expose network attachment, RF observation, and service endpoint endpoints.

Implementation references (`WISACS-API-005`): routes.py, schemas.py, wisacs_probe.py.

Verification references (`WISACS-API-005`): test_api.py, test_wisacs_probe.py.

7.5.6 WISACS-API-009: Interoperability Recommendations

The API MUST expose incident-scoped interoperability recommendations.

Implementation references (`WISACS-API-009`): routes.py, interop.py.

Verification references (`WISACS-API-009`): test_api.py.

7.5.7 WISACS-API-010: Demonstration Scenario Import

The API SHOULD provide a repeatable demonstration scenario import.

Implementation references (`WISACS-API-010`): routes.py, demo.py.

Verification references (`WISACS-API-010`): test_api.py.

7.5.8 WISACS-API-011: Incident Event Stream

The API SHOULD stream incident updates to subscribed clients over WebSockets.

Implementation references (`WISACS-API-011`): routes.py, events.py.

Verification references (`WISACS-API-011`): test_api.py.

7.5.9 WISACS-API-012: Draft SCA Descriptor Endpoint

The API MAY expose draft SCA-ready descriptors for field devices.

Implementation references (`WISACS-API-012`): routes.py, sca.py.

Verification references (`WISACS-API-012`): test_api.py.

7.6 API Shape

All operational REST endpoints are under `/api/v1`. The OpenAPI document is the executable API reference for V1.

7.7 Informative Error Handling

The prototype uses normal HTTP status codes: `401` for invalid or missing credentials, `403` for insufficient role authorization, `404` for missing resources, and

422 for schema validation errors generated by FastAPI/Pydantic. Future versions should standardize a structured error body once client behavior depends on machine-readable error categories.

8 WiSACS Data Model Specification

8.1 Purpose

The WiSACS data model preserves the minimum shared incident state needed to understand who is present, which communication systems are present, and which interoperability options can be reviewed. The model favors explicit incident scoping and flexible metadata fields because early disaster-response observations are incomplete and evolve quickly.

8.2 Entity Overview

Incident The root entity for an emergency-scene dataset.
Responder A person or operational unit associated with an incident.
Field device Equipment associated with a responder or incident.
Network A detected or manually entered RF/IP/voice/mesh/satellite communication system.
Network attachment A relationship between a device and a network with reachability evidence.
RF observation A spectrum observation produced manually, by a simulator, or by future sensor ingestion.
Service endpoint A reachable service exposed by a device on a network.
Interop recommendation A computed advisory record with evidence and risk metadata.
Event log A lightweight audit/event record for incident state changes.

8.3 Relationship Model

Every operational entity is scoped to an incident. Devices may be assigned to responders, may attach to one or more networks, and may expose service endpoints. Networks may have many device attachments and may carry service endpoints. RF observations may be sourced by a device when the observer is known.

The V1 schema stores latitude and longitude as scalar fields for broad client compatibility. Operational areas and coverage areas are represented as GeoJSON-compatible JSON so the API can support maps without requiring every deployment to use spatial queries immediately.

8.4 Normative Requirements

8.4.1 WISACS-DATA-001: Core Entities

The data model MUST include incident, responder, field device, network, network attachment, RF observation, service endpoint, interop recommendation, and event log entities.

Implementation references (`WISACS-DATA-001`): models.py, schemas.py.

Verification references (`WISACS-DATA-001`): test_api.py.

8.4.2 WISACS-DATA-002: Location And Area Fields

Location-bearing entities MUST include latitude and longitude, and operational/coverage areas SHOULD be represented as GeoJSON-compatible metadata.

Implementation references (`WISACS-DATA-002`): models.py, demo.py, pages.py, WiSACSOldMontrealResponse.gd, main.dart.

Verification references (`WISACS-DATA-002`): test_api.py.

8.4.3 WISACS-DATA-003: Flexible Metadata

Entities SHOULD include JSON metadata fields so ontology tags and newly discovered attributes can be preserved without a schema migration.

Implementation references (`WISACS-DATA-003`): models.py, schemas.py.

Verification references (`WISACS-DATA-003`): test_api.py.

8.4.4 WISACS-DATA-004: Heterogeneous Demonstration Data

The demonstration data MUST contain enough heterogeneous device and network observations to produce at least three interoperability recommendations.

Implementation references (`WISACS-DATA-004`): demo.py, interop.py, wisacs_probe.py.

Verification references (`WISACS-DATA-004`): test_api.py, test_wisacs_probe.py.

8.4.5 WISACS-DATA-005: Startup Schema Reconciliation

The backend SHOULD reconcile the live database schema against the repository data model at startup by creating missing tables and adding missing columns, while avoiding destructive schema changes without explicit migration review.

Implementation references (`WISACS-DATA-005`): session.py, main.py.

Verification references (`WISACS-DATA-005`): test_schema_reconciliation.py.

8.5 Storage

PostgreSQL with PostGIS is the production datastore. SQLite is permitted as a local development and automated-test fallback.

Startup reconciliation is additive. It is intended to keep prototype deployments usable when the repository adds a table or column. Renames, drops, type changes, and stricter constraints require explicit Alembic migrations or other reviewed database operations.

8.6 Informative Metadata Guidance

The `metadata_tags`, `properties`, and descriptor metadata fields are intended for ontology tags, source-system hints, confidence annotations, and newly discovered attributes. These fields should not become a dumping ground for stable fields that clients depend on; stable cross-client behavior should graduate into named schema fields in later versions.

9 WiSACS Security Specification

9.1 Purpose

WiSACS V1 handles operationally sensitive situational-awareness data. Even in prototype form, the API distinguishes public health checks from authenticated operational behavior and avoids taking automatic live-network actions.

9.2 Security Boundary

The V1 backend is the trust boundary for incident state. Clients and probe tools are treated as untrusted input sources. The backend validates request schemas, authenticates users, applies role checks to privileged actions, and records state through the database model.

9.3 Threat Model

The prototype is primarily concerned with:

- unauthenticated access to incident inventory,
- unauthorized loading or mutation of incident state,
- browser-delivered visualization code that shares an origin with the API,
- accidental interpretation of advisory recommendations as automatic approval,
- accidental claims of formal SCA compliance,
- and stale or fabricated observations being treated as field truth without review.

The prototype does not yet address production-grade federation, hardware attestation, incident-specific access policy, encrypted offline synchronization, or regulated audit retention.

9.4 Normative Requirements

9.4.1 WISACS-SEC-001: Bearer-Token Protection

Operational API endpoints MUST require bearer-token authentication unless explicitly documented as public health checks.

Implementation references (`WISACS-SEC-001`): deps.py, routes.py, security.py.

Verification references (`WISACS-SEC-001`): test_api.py.

9.4.2 WISACS-SEC-002: Privileged Setup Authorization

Privileged setup actions SHOULD require commander or administrator authorization.

Implementation references (`WISACS-SEC-002`): deps.py, routes.py.

Verification references (`WISACS-SEC-002`): test_api.py.

9.4.3 WISACS-SEC-003: No Automatic Bridging

WiSACS MUST NOT activate live network bridging or policy-changing actions without a later explicit authorization design.

Implementation references (`WISACS-SEC-003`): interop.py, routes.py.

Verification references (`WISACS-SEC-003`): test_api.py.

9.4.4 WISACS-SEC-004: Same-Origin Browser Surface

Browser-delivered WiSACS pages SHOULD call the operational API through same-origin paths and MUST NOT require permissive cross-origin access for the default public deployment.

Implementation references (`WISACS-SEC-004`): config.py, main.py, pages.py.

Verification references (`WISACS-SEC-004`): test_api.py.

9.5 V1 Security Posture

V1 uses local seeded demonstration users and JWT bearer tokens. Production federation, audit retention rules, and incident-specific access policy are future work.

9.6 Same-Origin Web Notes

The preferred public deployment places documentation, browser visualization, and API calls on the same HTTPS origin. This avoids making cross-origin API access part of the default security posture. Same-origin deployment does not remove the need for authentication, input validation, content security policy review, or production identity replacement.

9.7 Informative Authorization Notes

Authentication proves that a caller has a recognized prototype user identity. Authorization determines whether that caller can perform a privileged action. V1 uses simple role checks to keep the prototype reviewable. A production deployment should replace seeded users, static roles, and development secrets before use with real incident data.

10 WiSACS SCA Alignment Specification

10.1 Purpose

WiSACS V1 is SCA-ready rather than SCA-compliant. The prototype captures device and service metadata in a shape that can be reviewed by SCA practitioners and later transformed into formal descriptor artifacts if the workgroup chooses that direction.

10.2 Informative Alignment Goals

The V1 alignment work has three goals:

1. Preserve component-like metadata for field devices.
2. Represent ports and properties in a form that can be inspected outside the application.
3. Avoid overstating compliance before a dedicated conformance profile, descriptor format, and validation process exist.

10.3 Conceptual Mapping

WiSACS concept	SCA-adjacent concept	V1 treatment
Field device	Device or component	Stored with optional `sca_profile` and `component_kind`.
Capability	Component capability or property	Stored as structured JSON for later refinement.
Service endpoint	Port or service access point	Stored as protocol, port, URI, and status.

WiSACS concept	SCA-adjacent concept	V1 treatment
Device properties	Descriptor properties	Stored in the `properties` JSON field.
Descriptor metadata	Descriptor review context	Returned by the draft descriptor endpoint.

10.4 Normative Requirements

10.4.1 WISACS-SCA-001: SCA-Ready Device Metadata

Field devices SHOULD carry SCA-ready profile, component kind, port, property, and descriptor metadata fields where those concepts are known.

Implementation references (`WISACS-SCA-001`): models.py, schemas.py, demo.py.

Verification references (`WISACS-SCA-001`): test_api.py.

10.4.2 WISACS-SCA-002: SCA Descriptor Conformance Notice

Draft SCA descriptor output MUST clearly state that V1 descriptors are for review and are not a formal SCA conformance claim.

Implementation references (`WISACS-SCA-002`): sca.py, routes.py.

Verification references (`WISACS-SCA-002`): test_api.py.

10.5 Mapping

WiSACS field devices map loosely to SCA components or devices. Service endpoints map to exposed ports. Device properties and descriptor metadata preserve values that may later be transformed into formal SCA descriptor artifacts.

10.6 Informative Conformance Note

Reviewers should treat V1 descriptors as traceable metadata exports. They are useful for discussing SCA alignment, but they are not XML descriptor files, are not generated by an SCA toolchain, and are not validated against an SCA conformance suite.

11 WiSACS Render Deployment Specification

11.1 Purpose

The WiSACS deployment profile describes how the prototype can be hosted as a single Render-backed web application and managed PostgreSQL database. The public deployment uses one origin, `https://wisacs.winnforum.org`, for

documentation, browser visualization, and `/api/v1` API calls. The same project can run locally with SQLite for quick review or Docker Compose for a local PostgreSQL/PostGIS workflow.

11.2 Deployment Components

Web application service Runs the FastAPI application with Uvicorn, serves the documentation landing page, serves the lightweight HTML/SVG browser visualization, and exposes the operational API under `/api/v1`.

Installed clients Godot and future Flutter clients may run on handsets or desktops and call the same hosted API origin.

PostgreSQL database Stores incident, inventory, observation, recommendation, user, and event data.

PostGIS extension Provides the production geospatial foundation for future spatial queries and indexes.

11.3 Configuration Model

Render supplies `DATABASE_URL` from the managed database. The web application also reads `WISACS_SECRET_KEY`, `WISACS_ENV`, and `WISACS_PUBLIC_BASE_URL`. Local development may omit `DATABASE_URL`, in which case the application uses SQLite.

11.4 Normative Requirements

11.4.1 WISACS-DEPLOY-001: Render Blueprint

The repository SHOULD include a Render blueprint for one public web application service and one managed PostgreSQL database.

Implementation references (`WISACS-DEPLOY-001`): render.yaml, main.py.

Verification references (`WISACS-DEPLOY-001`): README.md, deployment-rend
er.md.

11.4.2 WISACS-DEPLOY-002: PostGIS Enablement

Production PostgreSQL deployments SHOULD enable PostGIS for geospatial indexing and future spatial queries.

Implementation references (`WISACS-DEPLOY-002`): docker-compose.yml, session.py, env.py.

Verification references (`WISACS-DEPLOY-002`): test_api.py.

11.4.3 WISACS-DEPLOY-003: Single-Origin Public Surface

The public WiSACS deployment MUST serve documentation at the origin root, browser visualization at `/visualization`, and operational APIs under `/api/v1`

on the same HTTPS origin.

Implementation references (`WISACS-DEPLOY-003`): render.yaml, main.py, pages.py.

Verification references (`WISACS-DEPLOY-003`): test_api.py, README.md.

11.5 Deployment Notes

The backend start command is `uv run uvicorn wisacs_api.main:app --host 0.0.0.0 --port $PORT`. Render supplies the database connection string through `DATABASE_URL`. The `render.yaml` blueprint attaches `wisacs.winnforum.org` to the web service, references the Render PostgreSQL resource as `wisacs-db`, and uses `/ready` as the health check path.

11.6 Informative Operations Notes

The V1 deployment profile is suitable for prototype review and demonstration. Before production use, operators should add managed secret rotation, database backups, stronger identity integration, TLS/domain review, incident data retention policy, operational monitoring, and a decision on whether demo credentials are disabled or isolated from operational incident data.

12 WiSACS Traceability Specification

12.1 Purpose

Traceability keeps the specification, code, and tests moving together. Each normative requirement has a stable ID, a nearby implementation reference in the specification, and at least one code or workflow reference that can be checked automatically.

12.2 Traceability Model

Normative requirement A requirement declared in `docs/spec/` with a stable `WISACS-*` ID and RFC-style requirement language.

Implementation reference A link from the specification to source code, configuration, or scripts that implement the requirement.

Verification reference A link from the specification to tests, build targets, or review workflows that exercise the requirement.

Code-side reference A requirement ID present in implementation, tests, scripts, or configuration so automated checks can detect stale IDs.

Review PDF A Pandoc/LaTeX PDF with line numbers intended for comment-based review.

12.3 Normative Requirements

12.3.1 WISACS-TRACE-001: Automated Traceability Check

The repository MUST provide a traceability check that reports normative requirement IDs missing from implementation or test references.

Implementation references (`WISACS-TRACE-001`): check_traceability.py, requirements.py.

Verification references (`WISACS-TRACE-001`): Makefile.

12.3.2 WISACS-TRACE-002: Review PDF Workflow

The repository SHOULD provide a Pandoc and LaTeX review-PDF workflow with line numbers.

Implementation references (`WISACS-TRACE-002`): Makefile, build_wisacs_spec_docs.py, review-lineno.tex.

Verification references (`WISACS-TRACE-002`): docs/README.md.

12.4 Review Practice

Normative requirement IDs are stable anchors. Each normative requirement in `docs/spec/` SHOULD include implementation references and verification references immediately below the statement. Review comments SHOULD cite IDs and line numbers from generated PDFs when available.

12.5 Informative Maintenance Workflow

When a requirement changes, update the normative statement, the implementation reference, the verification reference, and the related code-side ID references in the same change. The traceability checker is intentionally lightweight; it proves that anchors exist, but reviewers still need to assess whether the referenced code actually satisfies the requirement.

13 WiSACS API Implementation Map

This code-adjacent document maps the normative WiSACS requirements to the implementation in this repository. Requirement text lives in the Pandoc Markdown files under docs/spec; verification mapping lives in backend/tests/WISACS_API_COVERAGE.md.

The references intentionally use file and symbol names instead of line numbers so the map remains stable when nearby code formatting changes.

13.1 Implementation By Requirement

The implementation details are written as per-requirement entries instead of a wide table. This keeps PDF output readable when source paths and symbol names are long.

13.1.1 WISACS-SYS-001 Implementation

Requirement: WISACS-SYS-001

Feature area: Incident-scoped discovery inventory.

Implementation references: `Incident`, `Responder`, `FieldDevice`, `Network`, `RfObservation`, and `ServiceEndpoint` in models.py; inventory route handlers in routes.py; demo scenario creation in demo.py; probe payload generation in wisacs_probe.py.

Status: Implemented for the V1 Simulation Profile.

13.1.2 WISACS-SYS-002 Implementation

Requirement: WISACS-SYS-002

Feature area: Location-bearing incident display.

Implementation references: latitude, longitude, operational area, and coverage area fields in models.py; GeoJSON-compatible demo areas in demo.py; browser visualization and playback in pages.py; Godot visualization in WiSACSOldMontrealResponse.gd; Flutter map scaffold in main.dart.

Status: Implemented for map display and demo data.

13.1.3 WISACS-SYS-003 Implementation

Requirement: WISACS-SYS-003

Feature area: Advisory-only interoperability behavior.

Implementation references: recommendation construction in interop.py; read-only recommendation endpoint in routes.py.

Status: Implemented; no live network bridge is activated.

13.1.4 WISACS-API-001 Implementation

Requirement: WISACS-API-001

Feature area: Health and readiness endpoints.

Implementation references: root health/readiness handlers in main.py; API-prefixed health/readiness handlers in routes.py.

Status: Implemented.

13.1.5 WISACS-API-002 Implementation

Requirement: WISACS-API-002

Feature area: Authentication and current-user identity.

Implementation references: `login` and `me` in routes.py; `get_current_user` in deps.py; JWT and password helpers in security.py; seeded demo users in bootstrap.py.

Status: Implemented for the V1 Simulation Profile.

13.1.6 WISACS-API-003 Implementation

Requirement: WISACS-API-003

Feature area: Incident creation, listing, and retrieval.

Implementation references: incident route handlers in routes.py; `IncidentCreate` and `IncidentRead` in schemas.py.

Status: Implemented.

13.1.7 WISACS-API-004 Implementation

Requirement: WISACS-API-004

Feature area: Responder, field-device, and network inventory.

Implementation references: responder, device, and network route handlers in routes.py; corresponding Pydantic schemas in schemas.py.

Status: Implemented.

13.1.8 WISACS-API-005 Implementation

Requirement: WISACS-API-005

Feature area: Attachment, RF observation, and service endpoint ingestion.

Implementation references: attachment, RF observation, and service endpoint route handlers in routes.py; schema definitions in schemas.py; simulated payload generation in wisacs_probe.py.

Status: Implemented.

13.1.9 WISACS-API-009 Implementation

Requirement: WISACS-API-009

Feature area: Incident-scoped interoperability recommendations.

Implementation references: `compute_recommendations` in interop.py; `/interop/recommendations` route in routes.py.

Status: Implemented as advisory computation.

13.1.10 WISACS-API-010 Implementation

Requirement: WISACS-API-010

Feature area: Repeatable demonstration scenario.

Implementation references: `import_demo_scenario` and CLI entrypoint in demo.py; `/imports/demo-scenario` route in routes.py.

Status: Implemented.

13.1.11 WISACS-API-011 Implementation

Requirement: WISACS-API-011

Feature area: Incident WebSocket event stream.

Implementation references: `EventBroker` in events.py; `/ws/incidents/{incident_id}` route in routes.py.

Status: Implemented as in-process prototype fanout.

13.1.12 WISACS-API-012 Implementation

Requirement: WISACS-API-012

Feature area: Draft SCA-ready descriptors.

Implementation references: `build_sca_descriptor` in sca.py; `/sca/descriptors/{device_id}` route in routes.py.

Status: Implemented without SCA conformance claim.

13.1.13 WISACS-DATA-001 Implementation

Requirement: WISACS-DATA-001

Feature area: Core SQLAlchemy entities and API schemas.

Implementation references: entity classes in models.py; Pydantic API schemas in schemas.py.

Status: Implemented.

13.1.14 WISACS-DATA-002 Implementation

Requirement: WISACS-DATA-002

Feature area: Geospatial display fields.

Implementation references: scalar latitude/longitude and GeoJSON-compatible fields in models.py; `_geojson_circle_hint` in demo.py; browser map overlay

and scenario playback in pages.py; Godot map elements in WiSACSOldMontrealResponse.gd; Flutter map scaffold in main.dart.

Status: Implemented for V1 display and demo workflows.

13.1.15 WISACS-DATA-003 Implementation

Requirement: WISACS-DATA-003

Feature area: Flexible metadata and ontology tags.

Implementation references: `metadata_tags`, `properties`, and descriptor metadata fields in models.py; matching schema fields in schemas.py.

Status: Implemented.

13.1.16 WISACS-DATA-004 Implementation

Requirement: WISACS-DATA-004

Feature area: Heterogeneous demonstration data.

Implementation references: mixed-agency scenario data in demo.py; recommendation rules in interop.py; supplemental probe payload in wisacs_probe.py.

Status: Implemented.

13.1.17 WISACS-DATA-005 Implementation

Requirement: WISACS-DATA-005

Feature area: Additive startup schema reconciliation.

Implementation references: `create_schema` and `reconcile_schema` in session.py; startup invocation in main.py; Alembic scaffold for reviewed non-additive migrations in env.py.

Status: Implemented for missing tables and missing columns; destructive schema changes remain explicit migration work.

13.1.18 WISACS-SEC-001 Implementation

Requirement: WISACS-SEC-001

Feature area: Bearer-token protection.

Implementation references: `OAuth2PasswordBearer` and `get_current_user` in deps.py; route dependencies in routes.py; token helpers in security.py.

Status: Implemented for the V1 Simulation Profile.

13.1.19 WISACS-SEC-002 Implementation

Requirement: WISACS-SEC-002

Feature area: Commander/admin role gates.

Implementation references: `require_commander_or_admin` in deps.py; privileged route dependencies in routes.py.

Status: Implemented for demo import and incident creation.

13.1.20 WISACS-SEC-003 Implementation

Requirement: WISACS-SEC-003

Feature area: No automatic network bridging.

Implementation references: recommendation-only service behavior in interop.py; absence of bridge activation routes in routes.py.

Status: Implemented by omission and advisory-only behavior.

13.1.21 WISACS-SEC-004 Implementation

Requirement: WISACS-SEC-004

Feature area: Same-origin browser surface.

Implementation references: closed default `cors_origins` in config.py; conditional CORS middleware and public page routes in main.py; relative `/api/v1` browser API calls and page security headers in pages.py.

Status: Implemented for the default V1 public deployment.

13.1.22 WISACS-SCA-001 Implementation

Requirement: WISACS-SCA-001

Feature area: SCA-ready device metadata.

Implementation references: SCA-adjacent device fields in models.py; schema fields in schemas.py; populated gateway and SDR examples in demo.py.

Status: Implemented as review metadata.

13.1.23 WISACS-SCA-002 Implementation

Requirement: WISACS-SCA-002

Feature area: Non-conformance notice in draft descriptor output.

Implementation references: `conformanceNotice` in sca.py; descriptor route in routes.py.

Status: Implemented.

13.1.24 WISACS-DEPLOY-001 Implementation

Requirement: WISACS-DEPLOY-001

Feature area: Render deployment blueprint.

Implementation references: single web service and database definitions in render.yaml; FastAPI app startup and public routes in main.py.

Status: Implemented as a single-service deployment scaffold.

13.1.25 WISACS-DEPLOY-002 Implementation

Requirement: WISACS-DEPLOY-002

Feature area: PostGIS enablement.

Implementation references: local PostGIS image in docker-compose.yml; `create_schema` in session.py; migration environment setup in env.py.

Status: Implemented for local and deployment database setup.

13.1.26 WISACS-DEPLOY-003 Implementation

Requirement: WISACS-DEPLOY-003

Feature area: Single-origin documentation, visualization, and API surface.

Implementation references: custom domain, health check, and single web service in render.yaml; root, `/visualization`, and `/spec` routes in main.py; browser page generation in pages.py; hosted API override support in WiSACSOldMontrealResponse.gd.

Status: Implemented for the V1 public prototype topology.

13.1.27 WISACS-TRACE-001 Implementation

Requirement: WISACS-TRACE-001

Feature area: Requirement traceability checker.

Implementation references: check_traceability.py; code-side registry in requirements.py.

Status: Implemented.

13.1.28 WISACS-TRACE-002 Implementation

Requirement: WISACS-TRACE-002

Feature area: Pandoc and line-numbered review output.

Implementation references: build_wisacs_spec_docs.py; review header reviewlineno.tex; document targets in Makefile.

Status: Implemented using Drone-style Pandoc rendering.

14 WiSACS API Verification Coverage

This test-adjacent document maps WiSACS requirements to focused regression coverage. Requirement text lives in the Pandoc Markdown files under docs/spec, while implementation mapping lives in backend/WISACS_API.md.

Run the focused verification set with:

```
uv run pytest backend/tests tools/tests
python3 scripts/check_traceability.py
```

14.1 Coverage By Requirement

The coverage details are intentionally written as per-requirement entries instead of a wide table so PDF output can wrap long test identifiers naturally.

14.1.1 WISACS-SYS-001 Coverage

Requirement: WISACS-SYS-001

Verification references: `test_demo_import_populates_inventory_and_recommendations` in `test_api.py`; `test_probe_payload_contains_expected_objects` in test_wisacs_probe.py.

Coverage notes: Confirms incident-scoped inventory is populated by the demo and probe paths.

14.1.2 WISACS-SYS-002 Coverage

Requirement: WISACS-SYS-002

Verification references: `test_demo_import_populates_inventory_and_recommendations` and `test_single_origin_documentation_and_visualization_pages` in test_api.py.

Coverage notes: Confirms location-bearing demo entities are available through the API and the browser visualization route is served. Flutter map rendering remains a future Flutter-SDK verification item.

14.1.3 WISACS-SYS-003 Coverage

Requirement: WISACS-SYS-003

Verification references: recommendation assertions in `test_demo_import_populates_inventory_and_recomme`

Coverage notes: Confirms WiSACS produces advisory recommendations, not bridge activation actions.

14.1.4 WISACS-API-001 Coverage

Requirement: WISACS-API-001

Verification references: `test_health_and_ready` in test_api.py.

Coverage notes: Covers root and API-prefixed health/readiness endpoints.

14.1.5 WISACS-API-002 Coverage

Requirement: WISACS-API-002

Verification references: `test_auth_me` in test_api.py.

Coverage notes: Covers login and bearer-token current-user lookup.

14.1.6 WISACS-API-003 Coverage

Requirement: WISACS-API-003

Verification references: demo incident retrieval and inventory assertions in test_api.py.

Coverage notes: Covers incident availability after demo import; explicit create/list edge coverage is future hardening.

14.1.7 WISACS-API-004 Coverage

Requirement: WISACS-API-004

Verification references: responder, device, and network counts in `test_demo_import_populates_inventory_ar`

Coverage notes: Covers list endpoints through the demo import smoke path.

14.1.8 WISACS-API-005 Coverage

Requirement: WISACS-API-005

Verification references: probe payload test in test_wisacs_probe.py; demo import assertions in test_api.py.

Coverage notes: Covers generated payload shape and API availability through demo observations.

14.1.9 WISACS-API-009 Coverage

Requirement: WISACS-API-009

Verification references: recommendation type assertions in `test_demo_import_populates_inventory_and_rec`

Coverage notes: Covers shared-network, candidate-gateway, IP-relay, and voice-only recommendation types.

14.1.10 WISACS-API-010 Coverage

Requirement: WISACS-API-010

Verification references: demo import call in test_api.py; root demo smoke command documented in README.md.

Coverage notes: Covers repeatable import with reset behavior.

14.1.11 WISACS-API-011 Coverage

Requirement: WISACS-API-011

Verification references: `test_incident_websocket_connects` in test_api.py.

Coverage notes: Covers WebSocket connection and initial event. Multi-client fanout is future hardening.

14.1.12 WISACS-API-012 Coverage

Requirement: WISACS-API-012

Verification references: `test_sca_descriptor_for_gateway` in test_api.py.

Coverage notes: Covers descriptor metadata and non-conformance notice.

14.1.13 WISACS-DATA-001 Coverage

Requirement: WISACS-DATA-001

Verification references: backend API tests in test_api.py.

Coverage notes: Covers schema creation and entity use through FastAPI TestClient.

14.1.14 WISACS-DATA-002 Coverage

Requirement: WISACS-DATA-002

Verification references: demo import and inventory assertions in test_api.py.

Coverage notes: Confirms location fields are accepted and returned for demo data.

14.1.15 WISACS-DATA-003 Coverage

Requirement: WISACS-DATA-003

Verification references: schema and demo paths exercised in test_api.py.

Coverage notes: Metadata fields are exercised indirectly; explicit metadata mutation tests are future hardening.

14.1.16 WISACS-DATA-004 Coverage

Requirement: WISACS-DATA-004

Verification references: `test_demo_import_populates_inventory_and_recommendations`; `test_probe_payload_contains_expected_objects`.

Coverage notes: Confirms heterogeneous demo data yields at least three recommendation categories.

14.1.17 WISACS-DATA-005 Coverage

Requirement: WISACS-DATA-005

Verification references: `test_startup_schema_reconciliation_adds_missing_columns` in test_schema_reconciliation.py.

Coverage notes: Simulates a legacy database table and confirms startup reconciliation adds the missing model column.

14.1.18 WISACS-SEC-001 Coverage

Requirement: WISACS-SEC-001

Verification references: authenticated requests in test_api.py.

Coverage notes: Covers successful bearer-token path. Negative auth tests are future hardening.

14.1.19 WISACS-SEC-002 Coverage

Requirement: WISACS-SEC-002

Verification references: commander-authenticated demo import in test_api.py.

Coverage notes: Covers positive commander path. Role-rejection tests are future hardening.

14.1.20 WISACS-SEC-003 Coverage

Requirement: WISACS-SEC-003

Verification references: recommendation-only assertions in test_api.py.

Coverage notes: Confirms recommendations are advisory data records.

14.1.21 WISACS-SEC-004 Coverage

Requirement: WISACS-SEC-004

Verification references: `test_single_origin_documentation_and_visualization_pages` in test_api.py.

Coverage notes: Confirms the browser visualization is served by the API service and calls `/api/v1` through same-origin paths.

14.1.22 WISACS-SCA-001 Coverage

Requirement: WISACS-SCA-001

Verification references: `test_sca_descriptor_for_gateway` in test_api.py.

Coverage notes: Confirms SCA-ready metadata is present for the gateway demo device.

14.1.23 WISACS-SCA-002 Coverage

Requirement: WISACS-SCA-002

Verification references: `test_sca_descriptor_for_gateway` in test_api.py.

Coverage notes: Confirms descriptor output says it is not a formal SCA conformance claim.

14.1.24 WISACS-DEPLOY-001 Coverage

Requirement: WISACS-DEPLOY-001

Verification references: blueprint review in render.yaml.

Coverage notes: Deployment blueprint exists; live Render deployment is outside local regression scope.

14.1.25 WISACS-DEPLOY-002 Coverage

Requirement: WISACS-DEPLOY-002

Verification references: local schema creation exercised in test_api.py; PostGIS statements in session.py and env.py.

Coverage notes: SQLite tests cover app startup; PostGIS activation is configured but not exercised without a Postgres service.

14.1.26 WISACS-DEPLOY-003 Coverage

Requirement: WISACS-DEPLOY-003

Verification references: `test_single_origin_documentation_and_visualization_pages` in test_api.py; blueprint review in render.yaml.

Coverage notes: Confirms root documentation, `/visualization`, and `/spec` web routes are present locally; live custom-domain verification is a Render operations step.

14.1.27 WISACS-TRACE-001 Coverage

Requirement: WISACS-TRACE-001

Verification references: `python3 scripts/check_traceability.py` in the root Makefile.

Coverage notes: Traceability runs as part of `make test`.

14.1.28 WISACS-TRACE-002 Coverage

Requirement: WISACS-TRACE-002

Verification references: `make docs-review`; build_wisacs_spec_docs.py.

Coverage notes: Review PDF generation is verified by the documentation build workflow.