🛰️🚨 “SatLight”: Light the lab with corresponding colors when specific satellites pass overhead

A Loft Orbital Coding Challenge

By Christina Kneis Wolfenden

Context & Stakeholders

🖼️ Context

- Purpose
  - Enhance situational awareness in the office lab (lights as an ambient indicator of overhead satellite passes)
  - Could also serve as a demo/educational tool for visitors or staff
- Technical scope
  - Build a configurable Python utility that:
    - Reads a configuration file (satellites, colors, location, output options).
    - Queries an external satellite tracking API continuously.
    - Issues commands every 10 seconds to light hardware via configured outputs (STDOUT, file, or TCP).
- Constraints
  - Python: version ≥ 3.12, type annotations, pytest
  - Docker for containerization
  - Logging only to STDERR
  - Outputs limited to STDOUT, a file, or a TCP connection

👥 Stakeholders

- Loft (Client) Organization
  - Owns the office/lab where the satellite-pass lighting system will be deployed.
  - Cares about usability, reliability, and whether the system meets their operational need.
- End users (Loft staff in the lab)
  - People physically in the office who will see the lights changing.
  - They don’t care about code elegance; they care about “does the light reliably show when satellites are overhead?”
- Development and operations team (me)
  - Responsible for requirements gathering, design, implementation, and testing.
  - Must balance functionality with maintainability.
  - Will deploy, run, and troubleshoot the system after delivery.
  - Care about documentation, monitoring, and ease of updates.
- External APIs / Data Providers
  - They indirectly affect the system’s reliability.
  - Their uptime, rate limits, and API stability constrain what you can deliver.

Formal Constraint and Customer Need Definitions

🚧 Constraints (C)

- C-1: Python: version ≥ 3.12
- C-2: Python type annotations
- C-3: Pytest for unit tests
- C-4: Docker used for containerization
- C-5: Logging only to STDERR
- C-6: Outputs limited to STDOUT, a file, or a TCP connection

🙋 Customer Needs (CN)

- CN-1: Light the lab with specific colors when specific satellites pass overhead.
  - CN-1.1: Use a public satellite API.
  - CN-1.2: Application consumes a configuration file containing at least a list of satellites (each associated with a unique color) and a unique location for the lab (latitude and longitude)
  - CN-1.3: Emit a command every 10 s while one or more tracked satellites are overhead.
    - CN-1.3.1: Commands go to configurable outputs (STDOUT, file, TCP).
    - CN-1.3.2: Commands are in the form of NORAD_ID_0: color_0, NORAD_ID_1: color_1, .., NORAD_ID_N: color_N (e.g. 25544: blue, 48915: pink)
- CN-2: Service should be easy to install and interact with
  - CN-2.1: Service portability (Docker containerization)
    - CN-2.1.1: Dockerfile deliverable*
    - CN-2.1.2: docker-compose.yml deliverable*
  - CN-2.2: Service ease of interaction (Makefile, README.md)
    - CN-2.2.1: Makefile deliverable
    - CN-2.2.2: README.md deliverable that details the chosen solution and how to run it
- CN-3: High quality, tested, well-documented code
  - CN-3.1: Code visibility (documentation, comments)
  - CN-3.2: Code validation (unit tests)
  - CN-3.3: Code debugability (python typecasting)

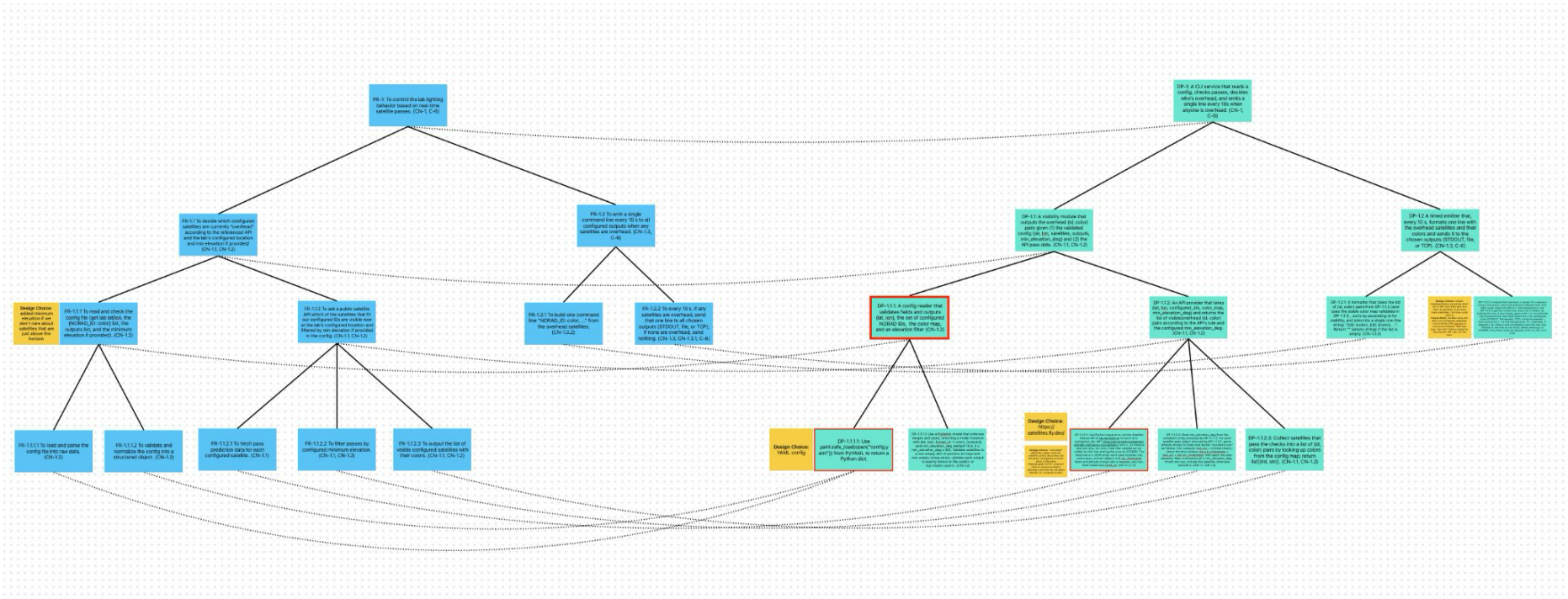
🎯 High Level Functional Requirements from C & CN

- FR-1: To control the lab lighting behavior based on real-time satellite passes. (CN-1, C-6)
- FR-2: To provide ease of installation and interaction for end user (CN-2, C-4)
- FR-3: To document, test, and easily debug codebase (CN-3, C-1, C-2, C-3, C-5)

FR-1 FR<>DP Decomposition

- FR-1: To control the lab lighting behavior based on real-time satellite passes. (CN-1, C-6)
  - FR-1.1 To decide which configured satellites are “overhead” according to the referenced API and the lab's location (CN-1.1, CN-1.2)
    - FR-1.1.1 To read and check the config file (get lab lat/lon, the {NORAD_ID: color} list, the outputs list, and the minimum elevation if provided). (CN-1.2)
      - FR-1.1.1.1 To load and parse the config file into raw data. (CN-1.2)
      - FR-1.1.1.2 To validate and normalize the config into a structured object. (CN-1.2)
    - FR-1.1.2 To ask a public satellite API which of the satellites that fit our configured IDs are visible now at the lab’s configured location and filtered by min elevation if provided in the config. (CN-1.1, CN-1.2)
      - FR-1.1.2.1 To fetch pass prediction data for each configured satellite. (CN-1.1)
      - FR-1.1.2.2 To filter passes by configured minimum elevation. (CN-1.1, CN-1.2)
      - FR-1.1.2.3 To output the list of visible configured satellites with their colors. (CN-1.1, CN-1.2)
  - FR-1.2 To emit a single command line every 10 s to all configured outputs when any satellites are overhead. (CN-1.3, C-6)
    - FR-1.2.1 To build one command line "NORAD_ID: color, ..." from the overhead satellites. (CN-1.3.2)
    - FR-1.2.2 To every 10 s, if any satellites are overhead, send that one line to all chosen outputs (STDOUT, file, or TCP); if none are overhead, send nothing. (CN-1.3, CN-1.3.1, C-6)
- DP-1: A CLI service that reads a config, checks passes, decides who’s overhead, and emits a single line every 10s when anyone is overhead. (CN-1, C-6)
  - DP-1.1: A visibility module that outputs overhead satellite (id, color) pairs given the config + API. (CN-1.1, CN-1.2)
    - DP-1.1.1: A config reader that validates fields and outputs (lat, lon), the set of configured NORAD IDs, the color map, and an elevation filter (CN-1.2)
      - DP-1.1.1.1: Use yaml.safe_load(open("config.yaml")) from PyYAML to return a Python dict.
      - DP-1.1.1.2 Use a Pydantic model that enforces ranges and types, returning a model instance with (lat, lon), {norad_id → color}, [outputs], and min_elevation_deg (default 10.0, 0 ≤ min_elevation_deg ≤ 90). Validate satellites is a non-empty dict of positive-int keys and non-empty string colors; validate each output is exactly stdout or file:<path> or tcp:<host>:<port>. (CN-1.2)
    - DP-1.1.2: An API provider that takes (lat, lon, configured_ids, color_map, min_elevation_deg) and returns the list of visible/overhead (id, color) pairs according to the API’s rule and the configured min_elevation_deg. (CN-1.1, CN-1.2)
      - DP-1.1.2.1: Use Python requests to call the Satellite Passes API at sat.terrestre.ar for each id in configured_ids: GET https://sat.terrestre.ar/passes/{id}?lat=<lat>&lon=<lon>&limit=1 with a ~5s timeout and one retry. On error, treat that satellite as not visible for this tick and log the error to STDERR. The response is a JSON array; each pass includes rise, culmination, and set objects with utc_timestamp fields and altitude strings (alt) in degrees, plus top-level visible and norad_id. (CN-1.1)
      - DP-1.1.2.2: Read min_elevation_deg from the validated config produced by DP-1.1.1.2. For each satellite pass object returned by DP-1.1.2.1, parse altitude strings to floats and decide “overhead now” as follows: first compute now_utc = int(time.time()); check the time window rise.utc_timestamp ≤ now_utc ≤ set.utc_timestamp; then apply the pass elevation filter culmination.alt ≥ min_elevation_deg. If both are true, include this satellite; otherwise exclude it. (CN-1.1, CN-1.2)
      - DP-1.1.2.3: Collect satellites that pass the checks into a list of (id, color) pairs by looking up colors from the config map; return list[(int, str)]. (CN-1.1, CN-1.2)
  - DP-1.2 A timed emitter that, every 10 s, formats one line with the overhead satellites and their colors and sends it to the chosen outputs (STDOUT, file, or TCP). (CN-1.3, C-6)
    - DP-1.2.1: A formatter that takes the list of (id, color) pairs from DP-1.1.2 (and uses the stable color map validated in DP-1.1.1) , sorts by ascending id for stability, and joins into a single one-line string: "{id}: {color}, {id}: {color}, ...". Return "" (empty string) if the list is empty. (CN-1.3.2)
    - DP-1.2.2: A module that maintains a steady 10 s cadence using a monotonic clock and subtracts elapsed work time each cycle: start tick with t0 = time.monotonic(); call DP-1.1.2 to get the current (id, color) list; if empty, do nothing this tick; if non-empty, pass to DP-1.2.1 to build the one command line, then send that line to the configured sinks (STDOUT, file append, TCP); compute elapsed = time.monotonic() - t0 and sleep(max(0, 10 - elapsed)). If elapsed ≥ 10, sleep 0 and immediately start the next tick. Failures in one sink do not block others; errors go to STDERR. Only these sinks are allowed. (CN-1.3, CN-1.3.1, C-6)

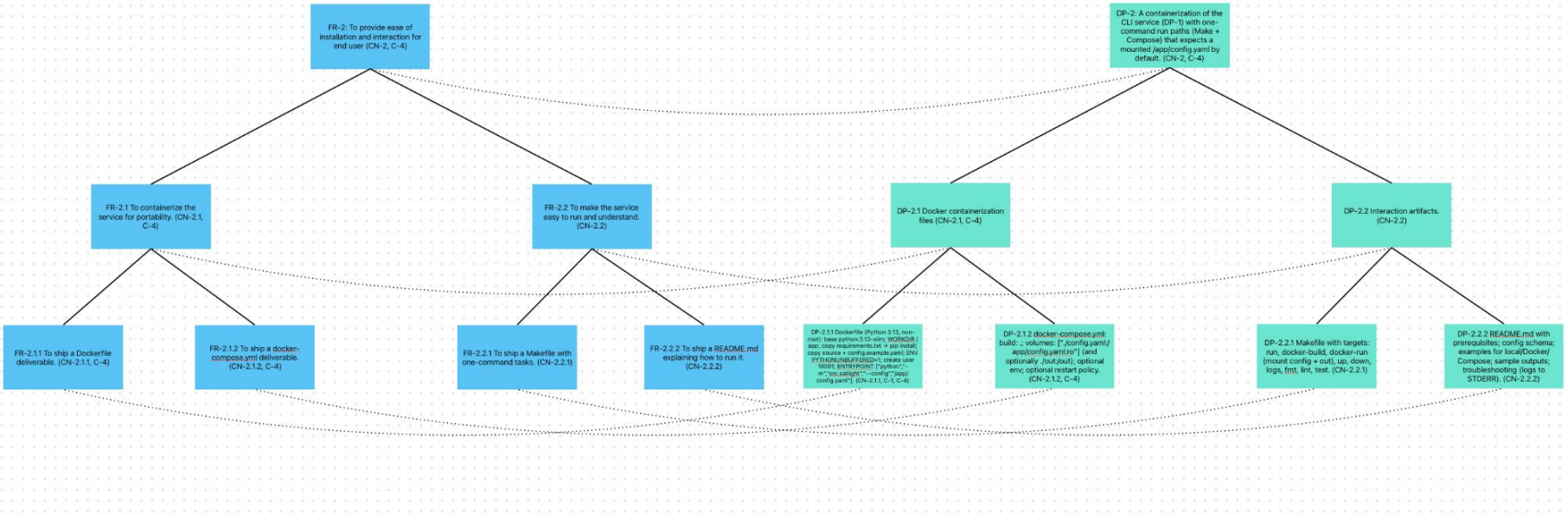
FR-1 FR<>DP Decomposition



⚙️ FR-2 FR<>DP Decomposition

- FR-2: To provide ease of installation and interaction for end user (CN-2, C-4)
  - FR-2.1 To containerize the service for portability. (CN-2.1, C-4)
    - FR-2.1.1 To provide a Dockerfile deliverable. (CN-2.1.1, C-4)
    - FR-2.1.2 To provide a docker-compose.yml deliverable. (CN-2.1.2, C-4)
  - FR-2.2 To make the service easy to run and understand. (CN-2.2)
    - FR-2.2.1 To ship a Makefile with one-command tasks. (CN-2.2.1)
    - FR-2.2.2 To ship a README.md explaining how to run it. (CN-2.2.2)
- DP-2: A containerization of the CLI service (DP-1) with one-command run paths (Make + Compose) that expects a mounted /app/config.yaml by default. (CN-2, C-4)
  - DP-2.1 Docker containerization files (CN-2.1, C-4)
    - DP-2.1.1 Dockerfile (Python 3.13, non-root): base python:3.13-slim; WORKDIR /app; copy requirements.txt → pip install; copy src/ and config.example.yaml; ENV PYTHONUNBUFFERED=1 PYTHONPATH=/app/src; create user 10001; ENTRYPOINT ["python","-m","satlight","--config","/app/config.yaml"]. (CN-2.1.1, C-1, C-4)
    - DP-2.1.2 docker-compose.yml: build: .; volumes: ["./config.yaml:/app/config.yaml:ro"] (and optionally ./out:/out); optional env; optional restart policy. (CN-2.1.2, C-4)
  - DP-2.2 Interaction artifacts. (CN-2.2)
    - DP-2.2.1 Makefile with targets: run, docker-build, docker-run (mount config + out), up, down, logs, fmt, lint, test. (CN-2.2.1)
    - DP-2.2.2 README.md with prerequisites; config schema; examples for local/Docker/Compose; sample outputs; troubleshooting (logs to STDERR). (CN-2.2.2)

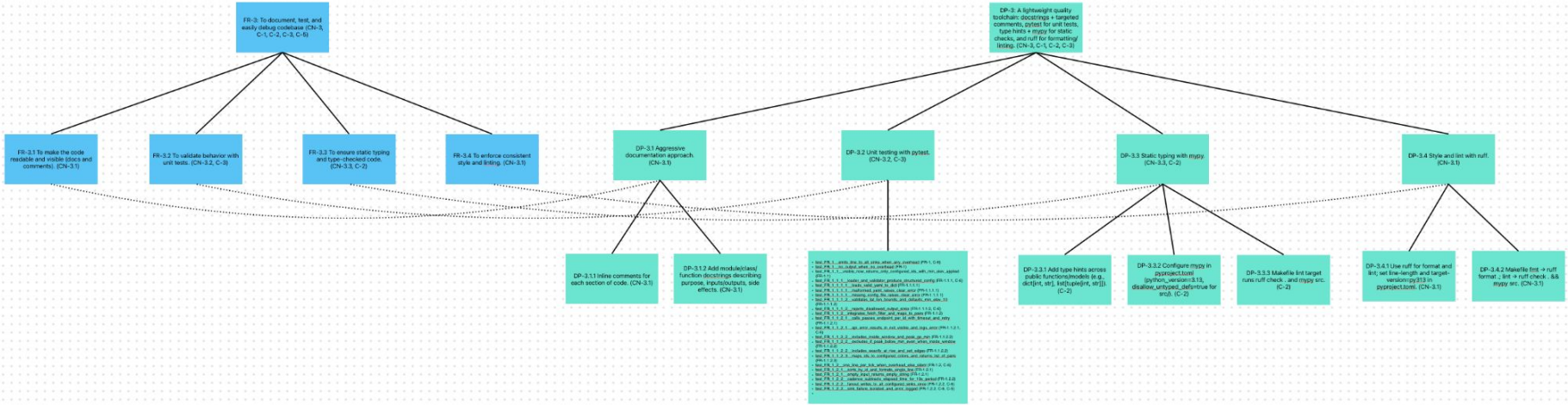
FR-2 FR<>DP Decomposition



⚙️ FR-3 FR<>DP Decomposition

- FR-3: To document, test, and easily debug codebase. (CN-3, C-1, C-2, C-3, C-5)
  - FR-3.1 To make the code readable and visible (docs and comments). (CN-3.1)
  - FR-3.2 To validate behavior with unit tests. (CN-3.2, C-3)
  - FR-3.3 To ensure static typing and type-checked code. (CN-3.3, C-2)
  - FR-3.4 To enforce consistent style and linting. (CN-3.1)
- DP-3: A lightweight quality toolchain: pytest for tests, type hints + mypy for static checks, ruff for format/lint, and a logging setup that writes only to STDERR. (CN-3, C-1, C-2, C-3)
  - DP-3.1 Aggressive documentation approach. (CN-3.1)
    - DP-3.1.1 Inline comments for each section of code. (CN-3.1)
    - DP-3.1.2 Add module/class/function docstrings describing purpose, inputs/outputs, side effects. (CN-3.1)
  - DP-3.2 Unit testing with pytest for all of FR-1 (see next). Use responses to mock HTTP and monkeypatch to simulate time (CN-3.2, C-3)
    -
  - DP-3.3 Static typing with mypy. (CN-3.3, C-2)
    - DP-3.3.1 Add type hints across public functions/models (e.g., dict[int, str], list[tuple[int, str]]). (C-2)
    - DP-3.3.2 Configure mypy in pyproject.toml (python_version=3.13, disallow_untyped_defs=true for src/). (C-2)
    - DP-3.3.3 Makefile lint target runs ruff check . and mypy src. (C-2)
  - DP-3.4 Style and lint with ruff. (CN-3.1)
    - DP-3.4.1 Use ruff for format and lint; set line-length and target-version=py313 in pyproject.toml. (CN-3.1)
    - DP-3.4.2 Makefile fmt → ruff format .; lint → ruff check . && mypy src. (CN-3.1)

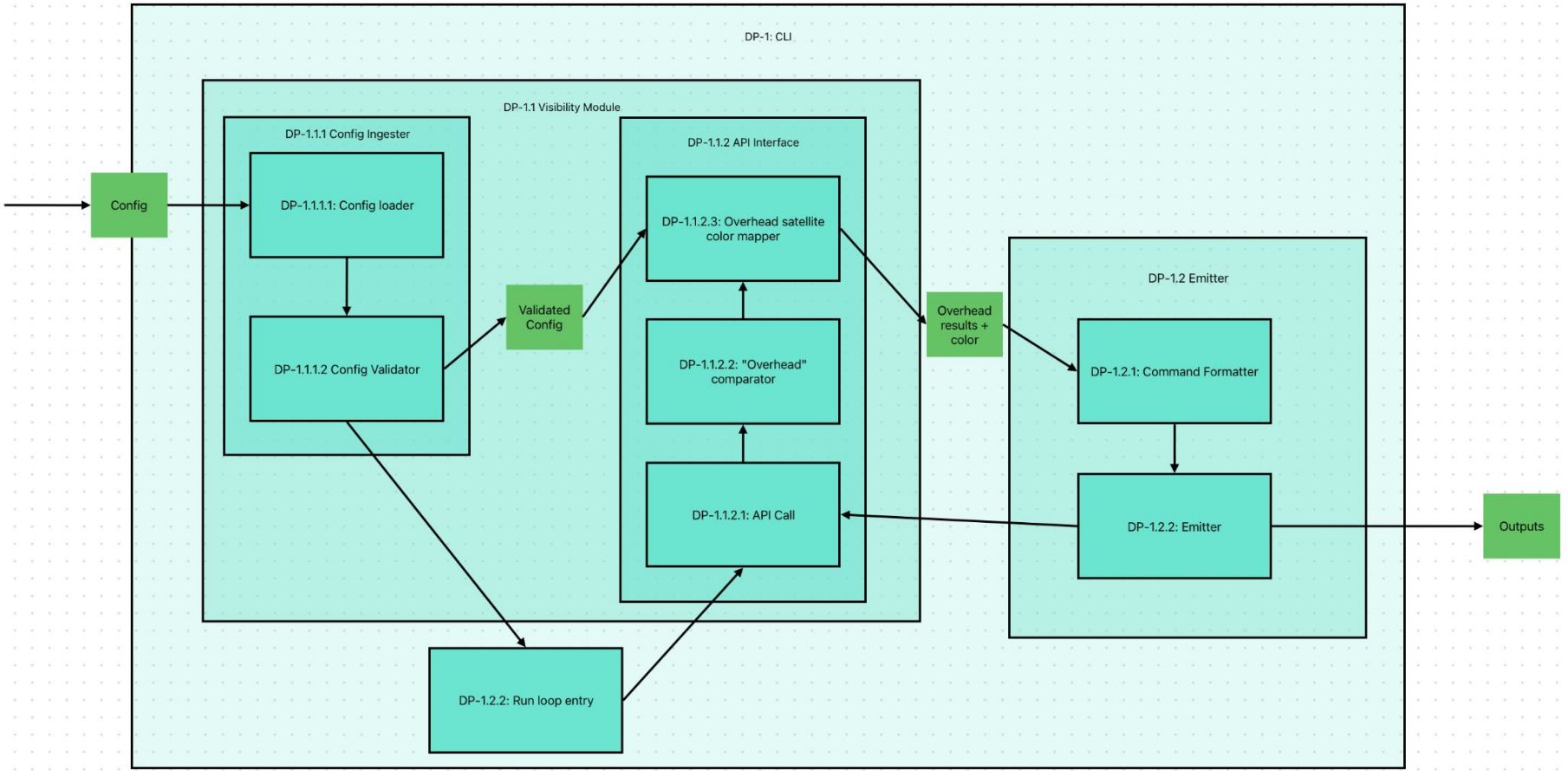
FR-3 FR<>DP Decomposition



🧪 Unit Test List

- test_FR_1__emits_line_to_all_sinks_when_any_overhead (FR-1, C-6)
- test_FR_1__no_output_when_no_overhead (FR-1)
- test_FR_1_1__visible_now_returns_only_configured_ids_with_min_elev_applied (FR-1.1)
- test_FR_1_1_1__loader_and_validator_produce_structured_config (FR-1.1.1, C-6)
- test_FR_1_1_1_1__loads_valid_yaml_to_dict (FR-1.1.1.1)
- test_FR_1_1_1_1__malformed_yaml_raises_clear_error (FR-1.1.1.1)
- test_FR_1_1_1_1__missing_config_file_raises_clear_error (FR-1.1.1.1)
- test_FR_1_1_1_2__validates_lat_lon_bounds_and_defaults_min_elev_10 (FR-1.1.1.2)
- test_FR_1_1_1_2__rejects_disallowed_output_sinks (FR-1.1.1.2, C-6)
- test_FR_1_1_2__integrates_fetch_filter_and_maps_to_pairs (FR-1.1.2)
- test_FR_1_1_2_1__calls_passes_endpoint_per_id_with_timeout_and_retry (FR-1.1.2.1)
- test_FR_1_1_2_1__api_error_results_in_not_visible_and_logs_error (FR-1.1.2.1, C-5)
- test_FR_1_1_2_2__includes_inside_window_and_peak_ge_min (FR-1.1.2.2)
- test_FR_1_1_2_2__excludes_if_peak_below_min_even_when_inside_window (FR-1.1.2.2)
- test_FR_1_1_2_2__includes_exactly_at_rise_and_set_edges (FR-1.1.2.2)
- test_FR_1_1_2_3__maps_ids_to_configured_colors_and_returns_list_of_pairs (FR-1.1.2.3)
- test_FR_1_2__one_line_per_tick_when_overhead_else_silent (FR-1.2, C-6)
- test_FR_1_2_1__sorts_by_id_and_formats_single_line (FR-1.2.1)
- test_FR_1_2_1__empty_input_returns_empty_string (FR-1.2.1)
- test_FR_1_2_2__cadence_subtracts_elapsed_time_for_10s_period (FR-1.2.2)
- test_FR_1_2_2__fanout_writes_to_all_configured_sinks_once (FR-1.2.2, C-6)
- test_FR_1_2_2__sink_failure_isolated_and_error_logged (FR-1.2.2, C-6, C-5)

System Architecture



🛠️ Implementation Overview

- Repo infrastructure and tooling
- Config ingest and validation
- API client
- Visibility analysis
- Output formatter
- Output emitter loop and sinks
- CLI
- Dockerize
- Patching + make better
- Documentation

🛠️ Implementation Step 1: Repo + Tooling

- What
  - Scaffold the project folders/files and wire up testing, linting, and typing so everything else has a solid base.
- Why
  - FR-3 / DP-3: documentation, tests, typing, linting = quality foundation.
  - C-1: Python ≥ 3.12 (we'll target 3.13).
  - C-2: type annotations → we add mypy.
  - C-3: unit tests → we add pytest.
  - (C-4/C-5 are addressed later; they don't block scaffolding.)
- How + tools
  - Layout: src/ package (satlight) + tests/unit/ for unit tests.
  - Config file: pyproject.toml to configure pytest, ruff, and mypy in one place.
  - Makefile: one-liners for fmt, lint, test, plus a simple local run.
  - .dockerignore: keeps containers lean later.

🛠️ Implementation Step 2: Config layer (load → validate)

- What
  - Implement:
    - DP-1.1.1.1: load_yaml(path) -> dict using PyYAML (safe loader).
    - DP-1.1.1.2: AppConfig (Pydantic v2) that validates and normalizes:
      - lat, lon
      - satellites: {NORAD_ID → color}
      - outputs: ["stdout" | "file:<path>" | "tcp:<host>:<port>"]
      - min_elevation_deg (default 10.0, bounds 0–90)
    - We'll also add unit tests for these.
- Why
  - FR-1.1.1: "read and check the config file": this is the ground truth for everything.
  - DP-1.1.1.1 + DP-1.1.1.2: exact DPs for load + validate/normalize.
  - CN-1.2: config-driven; we enforce the schema here.
  - C-6: allowed outputs are validated here so later code can assume they're safe.
- How + tools
  - PyYAML for safe YAML loading.
  - Pydantic v2 for strict validation + type normalization.
  - pytest for the unit tests listed.
  - We'll:
    - write load_yaml (wraps yaml.safe_load, friendly errors),
    - create AppConfig with validators (lat/lon bounds, satellites dict, outputs format, min_elevation_deg default/bounds),
    - expose validate_config(raw) -> AppConfig,
    - add tests for loader/validator.

🛠️ Implementation Step 3: API client

- What
  - Implement DP-1.1.2.1 in api.py:
    - fetch_next_pass(id, lat, lon, *, timeout=5) -> dict | None
    - Calls GET https://sat.terrestre.ar/passes/{id}?lat=<lat>&lon=<lon>&limit=1
    - On timeout: retry once; still failing → return None
    - On non-200 or bad JSON: return None
    - Log errors to STDERR (C-5) via our logger
    - Return the first pass object (dict) or None if no pass
  - We'll also write unit tests that use mock HTTP (responses) so tests are fast and deterministic.
- Why
  - FR-1.1.2.1: "fetch pass prediction data for each configured satellite" — this is that call.
  - DP-1.1.2.1: exact DP we're implementing.
  - CN-1.1: use a public satellite API.
  - C-5: errors go to STDERR (we log appropriately).
- How + tools
  - requests: simple HTTP client
  - responses: mock requests in tests (no network)
  - pytest: run tests, assert retry and error behavior
  - Our logger (src/satlight/log.get_logger) already routes to STDERR (C-5)

🛠️ Implementation Step 4: Visibility decision

- What
  - Implement DP-1.1.2.2 (filter rule) and DP-1.1.2.3 (collect (id,color) pairs) in visibility.py:
    - visible_now(cfg) will:
      - call fetch_next_pass for each configured NORAD ID,
      - compute now_utc = int(time.time()),
      - include a satellite iff rise.utc_timestamp ≤ now ≤ set.utc_timestamp and
      - culmination.alt ≥ cfg.min_elevation_deg,
      - return a list[(id, color)] (order unspecified; formatter handles sort).
  - We'll make it easy to test by allowing dependency injection for time and fetcher.
- Why
  - FR-1.1.2: ask the public API, apply min-elevation (from config), decide who's visible now.
  - DP-1.1.2.2/.3: this is exactly the filter + collect behavior.
  - CN-1.1 / CN-1.2: public API + config-driven decisions.
  - (Reminder: we're using culmination.alt as a proxy for pass elevation threshold: documented design choice.)
- How + tools
  - Python + standard library time.
  - We'll import AppConfig and reuse fetch_next_pass from Step 3.
  - Tests use monkeypatch to freeze time and pass in a fake fetcher (no real HTTP needed).

🛠️ Implementation Step 5: Output formatter

- What
  - Implement DP-1.2.1 in format.py:
    - format_line(pairs: list[tuple[int, str]]) -> str
    - Sort by ascending NORAD ID for stability
    - Join as: "25544: blue, 48915: pink"
    - Return "" if the input list is empty (sinks will then do nothing)
- Why
  - Maps to FR-1.2.1 (build the one command line), DP-1.2.1 (formatter), CN-1.3.2 (exact format the customer asked for).
  - Sorting ensures deterministic output so logs don’t jump around tick-to-tick.
- How + tools
  - Pure Python. No deps.
  - Keep it newline-free; sinks add \n. (Matches C-6 separation of responsibilities.)

🛠️ Implementation Step 6: Sinks + the 10-second emitter loop

- What
  - Implement C-6 sinks in sinks.py:
    - stdout_sink(line)
    - file_sink(path, line)
    - tcp_sink(host, port, line)
  - Implement DP-1.2.2 in emit.py:
    - A loop that every 10 s:
      - asks who's overhead (DP-1.1.2),
      - formats one line (DP-1.2.1),
      - fans out to sinks (C-6),
      - subtracts the time spent working from the 10 s period (monotonic clock),
      - logs errors to STDERR (C-5),
      - never lets one bad sink block others.
  - We'll also add unit tests for cadence, fanout, failure isolation, and the FR-1 top-level behavior ("emit when overhead / stay silent otherwise").
- Why
  - FR-1.2 / FR-1.2.2: emit a single command line every 10 s (or do nothing).
  - DP-1.2.2: the timed emitter, drift-free via monotonic time.
  - C-6: only STDOUT, file append, and TCP are allowed outputs.
  - C-5: all errors go to STDERR (our logger already does this).
- How
  - Plain Python (time.monotonic, time.sleep, socket, sys.stdout).
  - We inject time/sleep and visible_now via monkeypatch for deterministic tests.
  - We import the sinks module so we can monkeypatch those functions in tests.

🛠️ Implementation Step 7: CLI

- What
  - Implement a command-line entrypoint so a human can run:
    - python -m satlight.cli --config /path/config.yaml (loop forever), or
    - python -m satlight.cli --config /path/config.yaml --once (single tick, exits)
- Why
  - DP-1: the CLI service wrapper that runs the emitter.
  - FR-1 (top): real app behavior you can invoke.
  - C-5: all diagnostics/logs go to STDERR (only sinks write to STDOUT).
  - C-1/C-2/C-3: still using Python 3.13, typed functions, tested.
- How + tools
  - argparse for flags (--config, --once).
  - Our config layer load_yaml + validate_config (Step 2).
  - Our emitter run_forever | run_once (Step 6).
  - Logging via get_logger (to STDERR).

🛠️ Implementation Step 8: Dockerize

- What
  - Create:
    - Dockerfile (DP-2.1.1): Python 3.13 slim, non-root user 10001, PYTHONPATH=/app/src, ENTRYPOINT runs your CLI with default config path /app/config.yaml.
    - docker-compose.yml (DP-2.1.2): builds the image and mounts ./config.yaml → /app/config.yaml:ro (and ./out → /out).
    - requirements.txt (runtime deps).
    - config.example.yaml (so users know the schema).
    - Makefile targets for docker-build, docker-run, up, down, logs.
- Why
  - FR-2 (ease of install/interaction) → one-command flows.
  - DP-2 (containerization of DP-1) → reproducible runtime.
  - C-4 (Docker) honored: containerized Python 3.13.
  - Defaults to a mounted /app/config.yaml (what we've been planning).
- How + tools
  - Base image: python:3.13-slim.
  - Install only runtime deps (small image).
  - Non-root UID 10001 (safer).
  - ENTRYPOINT ["python","-m","satlight.cli","--config","/app/config.yaml"] —
  - extra args like --once can be appended to docker run / Compose.
  - Note: Docker installed on laptop is a prerequisite

🛠️ Implementation Step 9: Patch with pass cacheing

- What
  - Don’t re-call the API every tick. Cache the next pass window for each satellite and reuse it on every 10 s tick until it expires. Only refresh when:
    - you don’t have a pass cached, or
    - the cached pass ended (now > set time), or
    - you’re past a brief backoff after a failure.
    - This keeps your output cadence at 10 s, but drops API calls to rarely (typically once per satellite per pass).
  - Add a module-level round-robin index so each tick only fetches for one satellite (others use cache or skip).
  - Exponential backoff per satellite (with jitter) whenever a fetch fails (timeout/429/500/anything).
    - 60s → 120s → 240s → … capped at 3600s, with ±10% jitter.
    - This massively reduces repeated failures spamming the logs.
- Why
  - Keeps output cadence at 10 s (FR-1.2), but drops API calls to rarely
- How + tools
  - A patch that adds:
    - a per-satellite in-memory cache in visibility.py (valid until the pass’s set timestamp),
    - a 60 s backoff after a failed fetch (e.g., 429), so we don’t hammer the same sat.
    - add exponential backoff (with jitter) per satellite

🛠️ Implementation Step 10: Add a heartbeat for debugging

- What
  - Add a minimal heartbeat that logs once per tick to STDERR using existing _LOG. No config or env vars. No changes to sinks or FR-1.2.2 behavior.
- Why
  - Higher visibility into activities when there are no satellites overhead.
  - Maps to DP-1.2.2 (timed emitter) and C-5 (log to STDERR).
- How + tools
  - We'll patch emit.py to log once per tick.
  - We'll (re)define log.py to configure Python logging once, sending everything to STDERR, level driven by SATLIGHT_LOG_LEVEL (default INFO).
  - In Docker/Compose, we'll set SATLIGHT_LOG_LEVEL=DEBUG to see the heartbeat in make logs.

🛠️ Implementation Step 11: Linear interpolation for “overhead” bounds

- What
  - Add a patch to emit only while the pass is above the threshold right now.
- Why
  - Currently, ee emit from rise to set as long as the peak (culmination.alt) is above the threshold. That can start before the satellite has actually reached your min_elevation_deg.
- How + tools
  - Without changing APIs: Use the three points the API already gives you—rise (t, alt), culmination (t, alt), set (t, alt)—and assume the altitude changes linearly between them. Compute:
    - t_enter: when it first crosses up through min_elevation_deg (between rise → culmination)
    - t_exit: when it crosses down through the threshold (between culmination → set)
  - Then emit only if t_enter ≤ now ≤ t_exit (inclusive).

🛠️ Implementation Step 12: Documentation

- 📖 README.md Coverage:
  - Project Overview & Purpose
  - Core functionality: CLI service that emits satellite commands every 10s when satellites are overhead
    - Output format: NORAD_ID: color, NORAD_ID: color format
    - Design methodology: IDT decomposition (Customer Needs → Functional Requirements → Design Parameters)
  - Technical Architecture
    - How it works: 4-step process (config → API → overhead detection → emission)
    - Overhead definition: Uses peak elevation from pass windows, not instantaneous elevation
    - API politeness: Caching, round-robin fetching, exponential backoff with jitter
    - Timing: Drift-free 10s cadence using monotonic clock
  - Constraints Compliance
  - Design Choices & Tradeoffs (9 major areas)
  - User Guide
    - Configuration: Complete YAML schema with examples
    - Prerequisites: Python 3.13+, Docker, Make
    - Quick start: Both Docker and local development options
    - Makefile commands: 15+ commands for development, Docker ops, and output management
    - Sample outputs: STDOUT commands and STDERR diagnostics
    - Troubleshooting: Common issues and solutions
    - Architecture: Clean separation of concerns across 8 modules
- 📊 TRACEABILITY.md Coverage:
  - Functional Requirements Decomposition
  - Design Parameters Mapping
  - Testing Traceability Matrix
  - Evidence Collection

🔗 Traceability: FR-1 (Service Application)

Artifact	Verification method	Evidence (tests / checks)
FR-1 Control behavior by real-time passes	Unit tests (E2E)	`test_FR_1__emits_line_to_all_sinks_when_any_overhead` (also C-6), `test_FR_1__no_output_when_no_overhead`
FR-1.1 Decide which configured sats are “overhead”	Unit tests	`test_FR_1_1__visible_now_returns_only_configured_ids_with_min_elev_applied`
FR-1.1.1 Read & check config	Unit tests	`test_FR_1_1_1__loader_and_validator_produce_structured_config` (also enforces outputs; C-6)
FR-1.1.1.1Load/parse YAML	Unit tests	`test_FR_1_1_1_1__loads_valid_yaml_to_dict`, `test_FR_1_1_1_1__malformed_yaml_raises_clear_error`, `test_FR_1_1_1_1__missing_config_file_raises_clear_error`
FR-1.1.1.2Validate/normalize config	Unit tests	`test_FR_1_1_1_2__validates_lat_lon_bounds_and_defaults_min_elev_10`, `test_FR_1_1_1_2__rejects_disallowed_output_sinks` (C-6)
FR-1.1.2 Ask public API & filter	Unit tests	`test_FR_1_1_2__integrates_fetch_filter_and_maps_to_pairs`
FR-1.1.2.1 Fetch pass prediction per sat	Unit tests (HTTP mocked)	`test_FR_1_1_2_1__calls_passes_endpoint_per_id_with_timeout_and_retry`, `test_FR_1_1_2_1__api_error_results_in_not_visible_and_logs_error` (C-5)
FR-1.1.2.2 Filter by min elevation + time window	Unit tests	`test_FR_1_1_2_2__includes_inside_window_and_peak_ge_min`, `test_FR_1_1_2_2__excludes_if_peak_below_min_even_when_inside_window`, `test_FR_1_1_2_2__includes_exactly_at_rise_and_set_edges`
FR-1.1.2.3Output visible configured sats w/ colors	Unit tests	`test_FR_1_1_2_3__maps_ids_to_configured_colors_and_returns_list_of_pairs`
FR-1.2 Emit one line per 10 s when any are overhead	Unit tests	`test_FR_1_2__one_line_per_tick_when_overhead_else_silent` (C-6)
FR-1.2.1 Build `"NORAD_ID: color, ..."`	Unit tests	`test_FR_1_2_1__sorts_by_id_and_formats_single_line`, `test_FR_1_2_1__empty_input_returns_empty_string`
FR-1.2.2 10 s cadence & fan-out to sinks	Unit tests	`test_FR_1_2_2__cadence_subtracts_elapsed_time_for_10s_period`, `test_FR_1_2_2__fanout_writes_to_all_configured_sinks_once` (C-6), `test_FR_1_2_2__sink_failure_isolated_and_error_logged` (C-6, C-5)

🔗 Traceability: FR-2 (Install/Interaction)

Artifact	Verification method	Evidence (checks / commands)
FR-2.1.1 Dockerfile deliverable	Build succeeds	`docker build -t satlight:dev .` (also via `make docker-build`)
FR-2.1.2 docker-compose.yml deliverable	Compose up works	`make up` starts service; `make logs` shows 10 s heartbeat (STDERR)
FR-2.2.1 Makefile tasks	Commands exist & run	`make run / fmt / lint / test / docker-build / docker-run / up / down / logs`
FR-2.2.2 README.md	Manual review	README is present and explains config, run modes, outputs, troubleshooting

🔗 Traceability: FR-3 (Docs/Tests/Typing/Lint)

Artifact	Verification method	Evidence
FR-3.1 Docs & comments	Manual/code review	Docstrings and inline comments in modules; README present
FR-3.2 Unit tests	Automated	`make test` (CI: `pytest`) passes
FR-3.3 Static typing	Automated	`make lint` runs `mypy src` (passes); `pyproject.toml` sets `python_version=3.13`, `disallow_untyped_defs=true`
FR-3.4 Style & lint	Automated	`make fmt` (`ruff format`) and `make lint` (`ruff check`) pass cleanly

🔗 Traceability: Constraint Verification

Constraint	What it means here	Verification
C-1 Python ≥ 3.12	Project targets Python 3.13	Dockerfile uses `python:3.13-slim`; `pyproject.tomlpython_version=3.13`
C-2 Type annotations	Public functions/models typed; static checks	`mypy` passes via `make lint`; hints across code (`dict[int, str]`, `list[tuple[int,str]]`, etc.)
C-3 Pytest for unit tests	Tests exist and run	`make test` passes; CI runs `pytest`
C-4 Docker containerization	Shippable container & compose	Dockerfile + `docker-compose.yml`; `make docker-build`, `make up`succeed
C-5 Logs only to STDERR	No logs on STDOUT; errors/info on STDERR	Tests: `test_FR_1_1_2_1__api_error_results_in_not_visible_and_logs_error`, `test_FR_1_2_2__sink_failure_isolated_and_error_logged`; code routes logging to STDERR
C-6 Outputs limited (STDOUT/file/TCP)	Only these sinks; invalid rejected	Tests: `test_FR_1_1_1_2__rejects_disallowed_output_sinks`, `test_FR_1_2_2__fanout_writes_to_all_configured_sinks_once`; sinks implemented are `stdout`, `file`, `tcp` only

📋 Acceptance Checklist

- ✅ make fmt / make lint / make test → green
- ✅ python -m satlight.cli --config config.yaml --once (local) → runs without errors
- ✅ make docker-build && make docker-run → runs cleanly
- ✅ make up && make logs → shows heartbeat every ~10 s; emits lines during passes
- ✅ README.md explains config & limitations; config.example.yaml included
- ✅ TRACEABILITY.md committed

⚖️ Design Choices and Tradeoffs

- 1) Public API & “overhead” definition
  - Choice: Use **sat.terrestre.ar per-satellite** passes endpoint.
    - Why: Free, simple JSON, no API key; enough fields (rise/culmination/set).
    - Tradeoff: No “what’s overhead now for this location?” discovery; you must pre-choose IDs in config. Occasional 429/500s.
    - Alternatives (later): Add a one-shot helper using N2YO’s /above to discover IDs; or fully local propagation (Skyfield + TLEs) for zero external calls.
  - Choice: “Overhead now” = rise ≤ **now** ≤ set AND culmination.alt ≥ **min_elevation_deg**.
    - Why: Fast, stable, easy to explain/implement.
    - Tradeoff: Uses peak elevation to gate the whole pass; not the exact instantaneous elevation at now. Slightly permissive near the edges.
    - Alternatives (later): Compute instantaneous elevation per tick (requires a different API or local propagation).
- 2) Cadence & timing
  - Choice: Drift-free 10 s loop using time.monotonic(); **subtract work time each tick**.
    - Why: Predictable behavior; avoids clock jumps; meets CN-1.3.
    - Tradeoff: If a tick is slow (API slowness), you might have back-to-back ticks (sleep 0). That’s technically ok.
    - Alternatives (later): Cron/scheduler (less precise); async scheduling (more complexity, not needed here).
  - Choice: One satellite fetch per tick (**round-robin**) + **in-memory cache** of pass windows.
    - Why: Be polite to the free API; smooths calls over time; still converges.
    - Tradeoff: Freshness delay if you track many IDs (they’ll update across several ticks).
    - Alternatives (later): Bump per-tick budget, batch fetch (if an API supports it), or move to local predictions.

⚖️ Design Choices and Tradeoffs

- 3) Rate limits & resilience
  - Choice: Timeout (~5s), one retry, exponential backoff with jitter; failures count as “not visible this tick.”
    - Why: Keeps the loop healthy; errors don’t block others (DP-1.2.2).
    - Tradeoff: During API trouble you may miss a legitimate pass. Logs make it visible but we do not “compensate.”
    - Alternatives (later): Multi-provider fallback; longer cache; persistent cache across restarts.
- 4) Configuration & validation
  - Choice: **YAML + Pydantic model**; strict validation of sinks and ranges.
    - Why: Human-friendly config; strong schema gives clear errors up front.
    - Tradeoff: Slightly heavier dependency; you must keep schema/docs in sync.
    - Alternatives: JSON/TOML; environment-only config; dynamic reloads.
  - Choice: min_elevation_deg default 10.0°; enforce 0 ≤ min ≤ 90.
    - Why: Sensible default aligning with common horizon thresholds.
    - Tradeoff: Some labs may prefer a different default; you can set it in config.

⚖️ Design Choices and Tradeoffs (not as important)

- 5) Output & logging (constraints-driven)
  - Choice: Outputs limited to STDOUT, file, TCP (C-6). Logs only to STDERR (C-5).
    - Why: Exactly matches constraints; clean separation (data vs diagnostics).
    - Tradeoff: No fancy sinks (MQTT/HTTP POST/etc.). No structured log export.
  - Choice: One line only when any sat is overhead; otherwise silent on STDOUT.
    - Why: Matches FR-1.2.2 precisely and keeps downstream consumers simple.
    - Tradeoff: “Quiet” can look like “dead” if you don’t check STDERR heartbeat.
  - Choice: Stable formatting "id: color, id: color" sorted by ID.
    - Why: Deterministic, easy to parse, diff, and eyeball.
    - Tradeoff: No custom formatting (by design). Easy to extend later if needed.
- 6) Simplicity of runtime model
  - Choice: Straightforward synchronous Python; no threads/async.
    - Why: Fewer moving parts; easier tests; predictable timing.
    - Tradeoff: Not maximally parallel; per-tick budget limits throughput.
    - Alternatives (later): Async httpx, threadpool for IO, or multi-process workers.

⚖️ Design Choices and Tradeoffs (not as important)

- 7) Packaging & ops
  - Choice: Docker (python:3.13-slim), non-root user, Compose mount /app/config.yaml.
    - Why: Constraint-driven, reproducible, portable, easy one-command runs (CN-2/C-4).
    - Tradeoff: Slight image weight vs. pure host install; but worth it for portability.
    - Alternatives (later): Multi-stage build to slim further; publish to a registry.
  - Choice: Makefile targets for run/lint/test/docker/compose.
    - Why: Removes long CLI incantations.
    - Tradeoff: Make is an extra tool; but it's standard on dev machines/CI.
- 8) Testing & quality
  - Choice: pytest + HTTP mocking + time monkeypatch; mypy + ruff.
    - Why: Fast feedback; ensures constraints (typing, testing) are enforced.
    - Tradeoff: Some boilerplate in tests; but huge reliability gain.
- 9) Scope boundaries (being intentional about "not doing")
  - Not doing: Instantaneous elevation, sunlit/visibility optics, TLE management, discovery endpoints, fancy sinks, metrics/telemetry, persistent cache.
    - Why: Keep the MVP minimal and constraint-true; ship a robust core first.
    - What this buys: A clean, testable, explainable system that meets all FRs and Cs.

💡 Reflections and Future Work

Reflections:

- “Overhead” is a product decision
  - Is something that only reach 10 degrees above the horizon really “overhead”?
    - Built in min elevation into the config so user can decide
    - Interpolation made behavior match human expectation despite more realtime insight from the API
  - Lesson: define semantics in user language first (“when the light should actually turn on”), then pick math and API
- Rate limits are a hidden requirement
  - The free API shaped the architecture more than I expected (429/5xx drove caching and budgets).
  - Lesson: Treat the API’s limits like part of the spec. Be polite (throttle/backoff) and be resilient (don’t crash, don’t spam).
- Simplicity scales until it doesn’t
  - Single-threaded loop was easy to reason about and test; the cost is limited throughput per tick.
  - Lesson: http://satellites.fly.dev API was chosen for its simplicity, but its limits (per-ID polling, no discovery/instant elevation) actually made it harder/more time consuming to implement more “custom” or accurate features. Start simple, but choose tools that keep options open.

💡 Reflections and Future Work

Future Work:

- Improve satellite tracking info storage for speed and resilience
  - **Async fetching** would allow us to track a lot of satellites
    - Python asyncio could be used to
      - fetch several satellites in parallel
      - Fetch satellite info separately from the emitter loop
  - **Persist cache** across restarts
    - Save pass windows to a small JSON file; fewer cold-start API hits.
- Better “now” elevation
  - Try an API that returns **instantaneous elevation**, or add local orbit math (Skyfield + TLE cache) behind a flag for exactness
- GUI with hot-reload config
  - GUI (that could run locally or on a web page) that:
    - Allows you to update the satellites you’re watching/colors/min elevation/etc. In a GUI and update in realtime
      - Watch config.yaml and re-load on change (or SIGHUP) so you can add satellites without restarting
      - Button to suggest “IDs over my location” using a paid API, then copy into config
    - Shows which satellites are overhead now, next passes, and the line we’re emitting.
- CI/CD & environments
  - GitHub Actions: lint/typecheck/tests → build Docker → push image. Add dev vs prod config/env files. (I’ve done this before but tbd if I will have time to implement before Thursday morning)

💡 Reflections and Future Work

Future Work:

- Improve satellite tracking info storage for speed and resilience
  - **Async fetching** would allow us to track a lot of satellites
    - Python asyncio could be used to
      - fetch several satellites in parallel
      - Fetch satellite info separately from the emitter loop
  - **Persist cache** across restarts
    - Save pass windows to a small JSON file; fewer cold-start API hits.
- Better “now” elevation
  - Try an API that returns **instantaneous elevation**, or add local orbit math (Skyfield + TLE cache) behind a flag for exactness
- GUI with hot-reload config
  - GUI (that could run locally or on a web page) that:
    - Allows you to update the satellites you’re watching/colors/min elevation/etc. In a GUI and update in realtime
      - Watch config.yaml and re-load on change (or SIGHUP) so you can add satellites without restarting
      - Button to suggest “IDs over my location” using a paid API, then copy into config
    - Shows which satellites are overhead now, next passes, and the line we’re emitting.
- CI/CD & environments
  - GitHub Actions: lint/typecheck/tests → build Docker → push image. Add dev vs prod config/env files. (I’ve done this before but tbd if I will have time to implement before Thursday morning)


SATELLITES