Smart Energy Dashboard | Milestone 2

Advanced Software Engineering (623.503, 25W)

2026-01-08

Group G

Ingrid Guza, Yuliia Lomonosova, Patrick James Malapit,
Sabrina Muhrer, Naga Pranusha Munjuluru, Walter Telsnig

Smart Energy Dashboard — Final Architecture & Quality Overview

A Home Energy Management–oriented dashboard to serve and **visualize photovoltaic and storage time-series** data via a stable **REST API**. Focus of our project was a proper software architecture, quality, and evolvability rather than feature completeness or UI design.

Goal

Design and implement a maintainable, testable backend-centric system that:

- exposes energy data through a well-defined API
- supports rapid UI iteration
- can evolve safely as data sources, resolution, and logic change

Key Challenges (architectural)

- **Volatile domain**: changing data sources, resolutions, and analytics requirements
- **Stable contracts**: API and UI must not break as internals evolve
- **Quality under time pressure**: correctness, security, and performance must be measurable
- **Scope control**: avoid premature architectural complexity

Architectural stance

- This was not a UI project. The focus was on architecture and quality — how to design a system that survives change in a volatile energy domain
- **Modular monolith** as the **primary architecture** to ensure clear boundaries, fast feedback, and low operational overhead.

Agenda

- Core Stack – Design Rationale
- Quality gates & CI (SonarCloud, coverage)
- Performance testing outlook (Locust)
- Backup
  - Acceptance Test Scenarios
  - Alternative Architecture Exploration - Microservices

Core Stack – Design Rationale I

Language

Python (3.13)

- Rich ecosystem for data processing, web development, and rapid prototyping.

Backend (contract-first)

FastAPI + Pydantic

- Type-hinted endpoints + automatic OpenAPI = “living contract”; strict request/response validation; supports async I/O.
- High performance (async execution), automatic API documentation (Swagger UI), and strict typing with Pydantic (Type Hints).

Data management & Persistence

Docker/Compose

Database: PostgreSQL (running in Docker)

(CSV existed only as an adapter for Milestone 1 MVP, DB-backed was the production path.)

- Robust, ACID-compliant relational database for structured time-series and user data (Alternative would have been InfluxDB)

ORM: SQLAlchemy 2.0

- Modern, flexible ORM, schema evolution handled explicitly via Alembic migrations
- Deterministic, versioned database changes (CI/CD and DIP friendly)
- PostgreSQL drivers: psychopg2 / asyncpg (used via SQLAlchemy)

Frontend Framework

Streamlit

- Rapid development of data dashboards, declarative UI syntax, and excellent integration with Python data libraries (Pandas).
- UI consumes REST, does not own business logic
- *streamlit-echarts* for interactive, high-performance charts.

Authentication

JWT (JSON Web Tokens)

- Stateless, secure authentication flow suitable for modern REST APIs (python-jose, passlib for hashing).

Quality gates

Ruff|lint / mypy / pytest (incl. coverage)

- ruff: lint + formatting consistency which prevents trivial defects & style drift
- mypy: static typing which enforces module contracts, catches None/type misuse early
- pytest: regression safety → unit + integration tests (API/DB boundaries)

SonarCloud (using coverage from pytest –cov)

- Automated detection of bugs + smells; measurable maintainability; supports the required “metrics” focus.

Locust

- Load & Performance testing of REST endpoints under realistic usage patterns
- Validates non-functional requirements (latency, throughput)

Core Stack – Design Rationale II

Project Structure

The project follows a Modular Monolith architecture with strong influences from Clean Architecture (Hexagonal / Ports & Adapters).

```
/
├── app/                # Application Entry Points (FastAPI wiring)
├── modules/            # Domain Logic (The "Core")
│   ├── pv/             # PV Domain (Ports, Domain Models)
│   ├── battery/        # Battery Domain
│   └── ...
├── infra/              # Infrastructure / Adapters
│   ├── pv/             # CSV/DB implementation of PV repositories
│   ├── database.py     # DB Connection setup
│   └── models/         # SQLAlchemy Database Models
├── ui/                 # Presentation Layer (Streamlit)
│   ├── app.py          # Main UI Entry
│   └── pages/          # Dashboard Pages
├── core/               # Shared Kernel / Cross-cutting concerns
│   ├── settings.py     # Configuration management
│   └── security.py     # Auth utilities
└── tests/              # Automated Tests
```

modules/: Contains the business logic. This is where the core value lies, kept pure and independent of database or UI frameworks.

infra/: Contains the implementation details. Here we talk to the database, read CSVs, etc.

ui/: Contains the user interface. It consumes the logic/data but doesn't define business rules.

Separation of Concerns: We strictly separate Domain (logic), Infrastructure (data access), and Presentation (UI).

Dependency Inversion: High-level modules (modules/) do not depend on low-level modules (infra/). Both depend on abstractions (Ports).

Example: modules/pv/ports.py defines the PVRepositoryPort protocol. infra/pv/repository_csv.py implements it. The domain logic only knows about the Port, not the CSV or DB implementation.

SRP (Single Responsibility Principle): Each module and class has a single job. CSVPVRepository only reads CSVs. PVTimeSeries only holds data structure.

OCP (Open/Closed Principle): The system is open for extension but closed for modification. We added a Database implementation for PVRepositoryPort without changing the Domain logic that uses it.

DIP (Dependency Inversion Principle): As mentioned above, using Protocol to invert dependencies.

Quality Gates | SonarCloud and Locust (Performance Test)

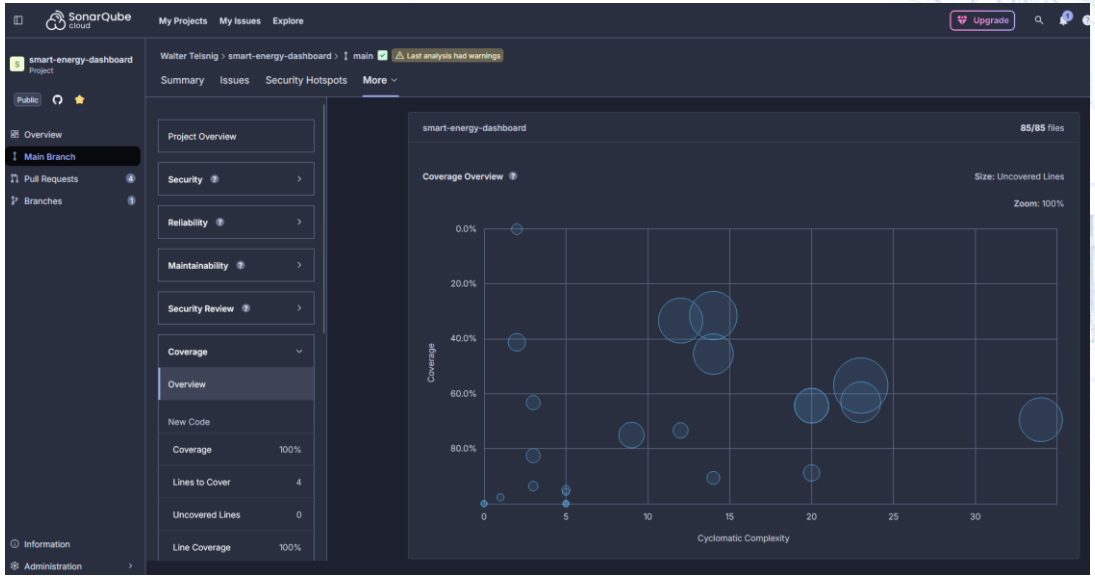
CI Quality Gates (Github Actions)

automated on every PR or change in main branch

SonarCloud (sonarcloud.io)

continuous code health, triggered by Github Actions

- Tracks **maintainability / code smells / reliability** issues over time
- Reports **test coverage** (trend-oriented)
- “Quality gate” : avoid architectural erosion via measurable signals
- Coverage rate: 76.8 %

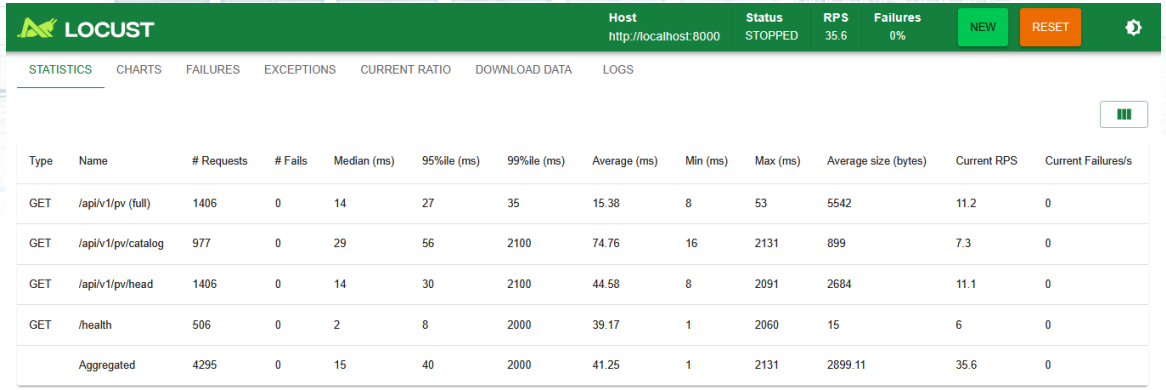


Performance Testing (Locust, headless)

On demand, focuses on API-level performance (UI – REST API – DB)

- Executed a headless Locust baseline test (10 users, ramp-up 2 users/s, 60 s runtime) against the REST API
- 356 requests, 0 failures, ~6 req/s sustained
- Typical latency is low: Aggregated p50 ≈ 12 ms, p95 ≈ 36 ms
- Steady-state endpoints are fast: /pv (full): p95 ≈ 17 ms, max ≈ 47 ms
- Tail latency observed on /pv/catalog and /pv/head (p99 ≈ 2–2.5 s), mainly due to warmup / occasional blocking operations
- Result: stable system under concurrent load, with clearly identified optimization candidates

The performance baseline shows excellent steady-state latency, while explicitly revealing cold-start and warmup effects. This allows us to distinguish optimization targets instead of relying on averages.



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/v1/pv (full)	1406	0	14	27	35	15.38	8	53	5542	11.2	0
GET	/api/v1/pv/catalog	977	0	29	56	2100	74.76	16	2131	899	7.3	0
GET	/api/v1/pv/head	1406	0	14	30	2100	44.58	8	2091	2684	11.1	0
GET	/health	506	0	2	8	2000	39.17	1	2060	15	6	0
	Aggregated	4295	0	15	40	2000	41.25	1	2131	2899.11	35.6	0

Alternative Architecture Exploration (Appendix)

What we explored

- Implemented a minimal microservice MVP for comparison and experimentation
- Focus: deployment & communication feasibility, not feature parity

Why it is not the main architecture

- Operational overhead (deployment, orchestration, monitoring)
- Higher load under tight project constraints
- Less learning value for core architectural principles at this stage

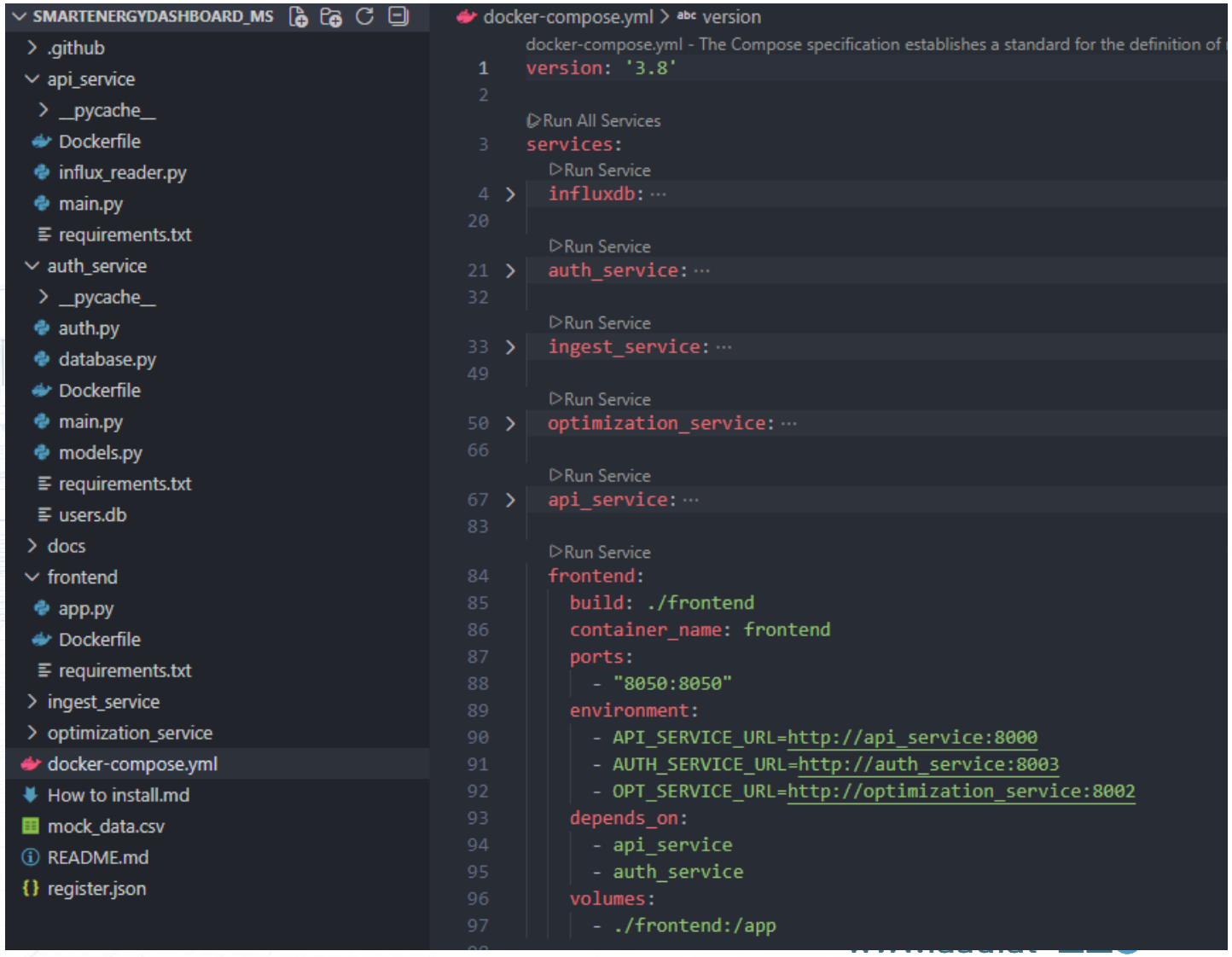
What we learned

Our modular monolith already provides:

- clear module boundaries
- service-like separation at code level
- Future service extraction is feasible without redesign

Conclusion

Microservices are a deployment decision, not a prerequisite for good architecture



Acceptance Test Scenarios

Runtime View – Read vs. Write

READ Path — PV time-series

Scenario: “Dashboard loads PV catalog / dataset”

1. Streamlit triggers GET /api/v1/pv/catalog
2. FastAPI router validates query + selects endpoint handler
3. PV module use case executes (no DB details here)
4. PVRepositoryPort is called (domain depends on interface)
5. DB adapter fetches from PostgreSQL via SQLAlchemy
6. Domain/DTO mapping → JSON response
7. Streamlit renders charts

Primary goal: fast, stable, cacheable reads

WRITE Path — Account + JWT

Scenario: “User registers or logs in”

1. Streamlit triggers POST /api/v1/accounts or /auth/login
2. FastAPI performs request validation (Pydantic)
3. Accounts use case applies business rules (uniqueness check, pwd const)
4. AccountRepositoryPort → DB adapter (SQLAlchemy)
5. Transaction commit/rollback in PostgreSQL
6. Security layer hashes password + issues JWT
7. Response returns token → Streamlit stores session state

Primary goal: correctness + security + consistency

- ✅ **Single contract boundary**: UI talks only to REST API (no direct DB access)
- ✅ **DIP enforced**: modules depend on Ports; infra provides adapters
- ✅ **Change localization**: DB/CSV/external API changes remain inside adapters
- ✅ **Different quality attributes per path**: READ → latency/throughput WRITE → security/consistency/validation
- ✅ **Testability**: paths can be tested at unit + integration level