Vision Paper – Smart Energy Dashboard

Yuliia Lomonosova, Patrick James Malapit, Sabrina Muhrer, Naga Pranusha Munjuluru, Walter Telsnig
Universität Klagenfurt, Austria
Emails: {yulomonosova, pamalapit, sabrinamuh, n1munjulur, wtelsnig}@edu.aau.at

I. Short project description

A residential PV system paired with a home battery raises self-consumption by storing surplus daytime generation and discharging it when household demand exceeds PV output. Rather than exporting excess energy at low remuneration, the battery shifts it to evening and night hours, offsetting retail purchases. Time–shifting also enables peak shaving (lowering instantaneous grid draw) and can reduce demand-related charges where applicable. Beyond bill savings, the setup can improve resilience (short-outage backup if supported by the inverter) and reduce the effective carbon footprint by maximizing on-site use of locally generated renewable energy. Key design factors include battery sizing relative to daily load and PV surplus, round-trip efficiency and degradation, inverter limits, and local tariff/net-metering rules.

The Smart Energy Dashboard is a Python-based platform that visualizes, analyzes, and manages PV and energy-storage data in real time. It aggregates time-series data (e.g., hourly PV generation, price curves, battery usage) to support operational decision-making. Our goal is a maintainable, extensible, and testable prototype that adheres to modern software-architecture principles (SRP, OCP, ADP, SDP, SAP) from *Clean Architecture* [1]. We start with a modular monolith for rapid iteration and conceptual integrity, ensuring a working deliverable by the end of the course, with a clear evolution path toward a microservices-driven design if time and scope permit. This paper intentionally omits business features beyond the MVP and focuses on architectural boundaries, interfaces, and evolution paths consistent with the course goals.

This paper intentionally focuses on *architectural boundaries, interfaces, and evolution paths* rather than an exhaustive feature set.

II. Main Goals / Functionality (Epics)

We prioritize a small, reliable core that we can iterate on safely: ingestion and serving of energy time series, a minimal but representative account module, and concise visualizations/KPIs. Inputs are normalized via adapters (CSV, database-backed, or semi–real-time streams) and published behind a *versioned, stable REST API*. Each epic corresponds to a cohesive module in the modular monolith with explicit interfaces, clear "definition of done," backward-compatible contracts, grouping changes by reason (CCP) and keeping an acyclic, stable dependency flow (ADP/SDP/SAP). The external API targets *Richardson Level 2* [2] (resource URIs + HTTP verbs) with path or header versioning for compatibility.

TABLE I
Main Goals / Functionality (Epics)

Epic Name	Description / User Goal
Account Management	Minimal account module (CRUD, authentication-ready boundary).
PV Data Integration	Ingest, catalog, and serve PV/consumption time series via REST.
Visualization Dashboard	Simple, trustworthy charts for production, consumption, and pricing.
Data Analytics Module	KPIs (e.g., self-consumption ratio, forecast accuracy), basic reports.
CI Pipeline	Automated tests, linting, and build (GitHub Actions).

III. Technical Plan (Tools & Rationale)

We follow a *Monolith First* [3] approach to optimize feedback cycles, simplify operations, and maintain conceptual integrity. Modules (Accounts, PV, Analytics) encapsulate a single responsibility (SRP), are open for extension (OCP), and depend on abstractions (DIP). Architectural decisions are recorded as ADRs (date, decision, status, consequences) to make rationale and trade-offs explicit and auditable.

FastAPI. We expose a versioned REST surface with type-hinted endpoints, automatic OpenAPI generation, and request/response validation. Lightweight dependency injection tightens contracts at module boundaries and supports async I/O if semi–real-time ingestion is required. OpenAPI artefacts act as living contracts and are versioned alongside code.

SQLAlchemy (Core/ORM). SQLAlchemy provides a mature abstraction over PostgreSQL with separation of concerns (schema models, queries, sessions), enabling repository or unit-of-work patterns. This aligns with DIP and supports CCP by localizing schema-related change.

Alembic. Alembic supplies reproducible, versioned database migrations. Coupling schema evolution to code revisions preserves closure around change (CCP), keeps deployments deterministic, and enables safe roll-forward/rollback across environments.

Quality and Operations. CI (GitHub Actions), tests (pytest), and linting/typing (ruff/mypy) function as quality gates that reduce integration risk and keep coupling low. Docker/Compose ensures environment parity and repeatable build/deploy surfaces.

This design yields: (i) *high cohesion/low coupling* via explicit interfaces and minimal cross-module dependencies; (ii) *stable contracts* at API and persistence boundaries; and (iii) an

evolution path where module boundaries can map to services later, consistent with ADP/SDP/SAP. In 4+1 terms, this paper focuses on the *Development view* (module/code structure) and *Process view* (runtime/API boundaries); the Logical/Physical views follow from these decisions.

TABLE II
TECHNICAL PLAN

Tool / Framework	**Purpose / Justification**
Python 3.13 + FastAPI	Modular monolith REST API; clear endpoints; fast dev; DI patterns.
SQLAlchemy + Alembic (PostgreSQL)	ORM and migrations; clean schema evolution (encourages OCP/CCP).
Docker + Docker Compose	Reproducible local/dev; smooth future microservice split if needed.
GitHub Actions + Pytest + Ruff	CI with tests & linting for maintainability and quality gates.
MkDocs / ADR Templates	Lightweight docs; record 4+1 views and architectural decisions [4].
Future: Streamlit UI	Optional interactive front-end for scenario exploration.

REFERENCES

[1] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Pearson, 2018.

[2] L. Richardson. (2008) Richardson maturity model. Accessed: 2025-10-20. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html

[3] M. Fowler. (2015) Monolithfirst. Accessed: 2025-10-20. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html

[4] P. Kruchten, "Architectural blueprints—the "4+1" view model of software architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.