EV Charging Station Simulator — Complete Documentation

This document consolidates the project overview, installation and run instructions, architecture, implemented features, API, dashboard, logging and monitoring, the smart charging policy engine, testing, integration notes, and recommended next steps.

Project root: /Users/nayankumar/Library/Mobile Documents/com~apple~CloudDocs/24version_3 2

```
Files of interest:
- [charging_policy.py](charging_policy.py)
- [station.py](station.py)
- [controller_api.py](controller_api.py)
- [csms_server.py](csms_server.py)
- [static/app.js](static/app.js)
- [static/styles.css](static/styles.css)
- [test_charging_policy.py](test_charging_policy.py)
- [readme.md](readme.md)
- [POLICY_ENGINE_ARCHITECTURE.md](POLICY_ENGINE_ARCHITECTURE.md)
- [POLICY_ENGINE_QUICK_REFERENCE.md](POLICY_ENGINE_QUICK_REFERENCE.md)
- [POLICY_ENGINE_IMPLEMENTATION.md](POLICY_ENGINE_IMPLEMENTATION.md)
- [MANIFEST.txt](MANIFEST.txt)
```

---

1. Summary

```
This repository contains a Python-based EV charging station swarm simulator (OCPP 1.6 JSON) with:
- A mock CSMS backend (`csms_server.py`) accepting standard OCPP messages
- A simulator engine for many virtual stations (`station.py`) with per-station behavior profiles
- A controller REST API + dashboard (`controller_api.py` + `static/*`) to control and monitor the swarm
- Per-station in-memory logs (50-entry circular buffer) exposed by `GET /stations/{station_id}/logs`
- A reusable, pure-function smart charging policy engine (`charging_policy.py`) with unit tests
- Comprehensive documentation and testing artifacts
```

This document explains how components interact, how to run the system locally, and where to find code and tests.

---

2. Installation

1. Create and activate a Python virtual environment:

```bash python -m venv venv source venv/bin/activate ```

2. Install dependencies:

```bash pip install -r requirements.txt ```

3. Start the CSMS backend (accepts station WebSocket connections):

```bash python csms_server.py ```

4. Start the controller API + UI (serves the dashboard at `/`):

```bash uvicorn controller_api:app --reload --port 8000 ```

Open the dashboard at: http://localhost:8000/

---

3. Project Structure & Key Files

- `station.py` — Simulator engine. Implements `SimulatedChargePoint`, per-station logging (deque, maxlen=50), transaction lifecycle, and integration points where smart charging and logging occur. It now uses the `charging_policy` engine.

- `controller_api.py` — FastAPI controller for managing stations: scale, start/stop, and new endpoint `GET /stations/{station_id}/logs` to retrieve last 50 events.

- `csms_server.py` — Minimal OCPP backend used during simulation/testing. Runs on port 9000 by default.

- `static/app.js` and `static/styles.css` — Dashboard frontend. Includes the station table, action buttons, collapsible per-station log viewer, caching, and UI styling.

- `charging_policy.py` — The pure policy engine. Exposes: - `evaluate_charging_policy(station_state, profile, env) -> {action, reason}` - `evaluate_meter_value_decision(...)` for meter-loop checks

- `test_charging_policy.py` — 24 unit tests covering energy cap, price threshold, peak hours, priorities, edge cases, and meter-loop behavior.

- Documentation files: `readme.md`, `POLICY_ENGINE_ARCHITECTURE.md`, `POLICY_ENGINE_QUICK_REFERENCE.md`, `POLICY_ENGINE_IMPLEMENTATION.md`, and `MANIFEST.txt`.

---

4. How the System Works (High Level)

```
1. The Controller UI sends commands to `controller_api.py` to scale/start/stop stations.
2. Each simulated station (`SimulatedChargePoint`) connects to the CSMS backend (`csms_server.py`) over Web
3. Before starting a transaction the station evaluates the policy engine to decide whether to `charge`, `wa
4. Stations log important events (startup, boot, auth success/fail, charging started/stopped, price blocks,
```

---

5. API Reference

```
Key REST endpoints provided by `controller_api.py`:
- `GET /` — Dashboard (serves index.html)
- `GET /stations` — List all simulated stations and their state
- `POST /stations/scale` — Scale the swarm
- `POST /stations/start` — Start a single station
- `POST /stations/stop` — Stop a single station
- `GET /stations/{station_id}/logs` — Retrieve up to 50 most recent log entries for a station (JSON: `{stat
- `GET /metrics` — Prometheus metrics endpoint
- `GET /pricing` and `POST /pricing` — Get/set simulated price
- `GET /totals` — Totals like energy and earnings
```

Refer to readme.md for examples and usage.

---

6. Dashboard & UI

```
- The dashboard shows: Stats Row (total stations, running, total energy, earnings), Scaling Controls, Price
- Each station row includes an "Actions" column with a "■ Logs" button. Clicking this expands a collapsibl
- The UI uses lazy-loading + smart in-memory caching for logs: first click fetches from API; subsequent tog
- Files: [static/app.js](static/app.js), [static/styles.css](static/styles.css), [templates/index.html](te
```

---

7. Per-Station Logging

```
- Implementation: Each `SimulatedChargePoint` maintains `log_buffer = deque(maxlen=50)`.
- Logging triggers include: station init, boot notifications, heartbeats, authorization results, transacti
- Logs are timestamped `[HH:MM:SS] message` and stored newest-last (UI displays newest first).
- Exposed via API: `GET /stations/{station_id}/logs`.
```

---

8. Smart Charging Policy Engine (Detailed)

File: charging_policy.py

Purpose: centralize decision-making so charging behavior is controlled by a pure function that is easy to test and extend.

Function signature (main):

```python evaluate_charging_policy(station_state: dict, profile: dict, env: dict) -> dict ```

```
Inputs:
- `station_state`: {"energy_dispensed": float (kWh), "charging": bool, "session_active": bool}
- `profile`: {"charge_if_price_below": float, "max_energy_kwh": float, "allow_peak_hours": bool, "peak_hour
- `env`: {"current_price": float, "hour": int}
```

Output (dict): - `action`: "charge" | "wait" | "pause" - `reason`: human-readable explanation (used by caller to log)

```
Decision order:
1. If energy_dispensed >= max_energy_kwh => `pause` ("Energy cap reached")
2. If current_price > charge_if_price_below => `wait` ("Price too high")
3. If hour in peak_hours and allow_peak_hours is False => `wait` ("Peak hour block")
4. Else => `charge` ("Conditions OK")
```

There is also `evaluate_meter_value_decision(...)` used inside the MeterValues loop to perform precise Wh-based checks and map results to `continue`/`stop` actions.

All logging is done by the caller (station code) based on returned `reason`.

---

9. Tests

```
- `test_charging_policy.py` contains 24 unit tests covering:
  - Energy cap boundary and exceed cases
  - Price threshold (above, below, at threshold)
  - Peak hour blocking behavior and multiple peak hours
  - Priority interactions (energy cap outranks price, price outranks peak)
  - Meter-loop decisions with precise Wh checks
  - Return structure verification
```

Run tests:

```bash python -m pytest test_charging_policy.py -v ```

All tests pass (24/24) in the current workspace.

---

10. Integration Notes (what changed in `station.py`)

```
- The old inline checks (`should_start_charging`, `get_energy_step_size`) were replaced by calls to the po
- Pre-transaction: `evaluate_charging_policy` is called with a fresh session state; if action != "charge" 
- During the MeterValues loop: `evaluate_meter_value_decision` is called with current accumulated Wh and ma
- `is_peak_hour()` helper remains for utility; heavy-lifting resides in `charging_policy.py`.
```

---

11. Deployment & Runtime Notes

```
- The system is designed to run locally for testing and development.
- For production-style load testing, run `uvicorn controller_api:app --port 8000` and scale stations from 
- The policy engine runs synchronously and is negligible cost (<1µs per call). Most overhead comes from OC
```

---

12. Next Steps & Enhancements

```
Suggested future work (already documented in architecture files):
- Real-time WebSocket log streaming for live log push
- Persistent log storage (DB) for long-term analysis
- Filtering/searching logs in UI
- Add ramping actions (e.g., `ramp_down`) to policy engine
- Grid-aware and carbon-aware constraints (add `env` fields like `grid_load`, `carbon_intensity`)
- Add CSV/JSON export for logs
- Add E2E tests for the UI/API
```

---

13. Contact & Handoff

If you need a tailored export (PDF, HTML) or an executive summary slide deck derived from this documentation, I can generate that next.

End of consolidated documentation. File location: COMPLETE_DOCUMENTATION.md