

# Siphyy Core Architecture Report

Generated from the local repository state on 2026-05-14.  
Intended as a shareable technical briefing for another model or engineer reviewing the project.

## 1. Executive Summary

Siphyy is an open-source Python framework for fleet telematics analytics, centered on a two-tier pipeline:

- Tier 1 deterministic detection:** provider telemetry is normalized into canonical events and passed through cheap, stateful rules that surface only suspicious candidates.
- Tier 2 LLM interpretation:** suspicious candidates are enriched with historical incident cases and sent to a pluggable LLM client that returns a structured report.

The current implementation is focused on one end-to-end use case: **fuel anomaly / siphonage detection**. The repo now contains a working reference adapter ( `TrakzeeAdapter` ), a working Tier 1 detector ( `FuelSiphonageDetector` ), a working Tier 2 agent ( `FuelAnomalyAgent` ), pluggable OpenAI and Anthropic clients, test coverage, and a hosted Streamlit demo that visualizes the pipeline and shows the exact LLM prompts used.

The project is no longer just a schema scaffold. It now demonstrates the complete path from uploaded telematics rows to a structured, auditable anomaly report.

## 2. High-Level Product Purpose

Fleet telematics providers all expose different payloads, field names, units, and data quality. Siphyy exists to provide a provider-agnostic analytics framework where:

- adapters isolate provider-specific quirks at the ingestion boundary,
- detectors operate only on canonical events,
- agents reason only on canonical evidence plus curated cases, and
- final outputs are structured and inspectable rather than opaque text.

The current value proposition is strongest in three areas:

- fuel theft / siphonage detection,
- false-positive suppression via Tier 2 case-grounded reasoning,
- demoability and transparency of the LLM call path.

## 3. Repository Layout

Path	Purpose
<code>src/siphyy/schema/</code>	Canonical event models, interesting-event models, case-base models.
<code>src/siphyy/adapters/</code>	Provider-specific translation into canonical events. Currently includes the Trakzee reference adapter.
<code>src/siphyy/detectors/</code>	Tier 1 deterministic detection layer and state-store abstraction.
<code>src/siphyy/agents/</code>	Tier 2 LLM agent layer plus provider-specific LLM client implementations.
<code>src/siphyy/knowledge/</code>	Seed incident cases used to ground the agent.
<code>examples/</code>	CLI-style quickstarts and small demos of the framework pipeline.
<code>apps/demo/</code>	Streamlit demo app for interactive uploads and step-by-step visualization.
<code>tests/</code>	Unit and integration tests for schemas, adapter, detector, agent, and LLM clients.
<code>.github/workflows/</code>	CI workflow and HuggingFace Spaces deployment workflow for the demo.

## 4. End-to-End Runtime Flow

```

[Provider rows / export]
↓
TrakzeeAdapter.adapt()
↓
Canonical TelemetryReading events
↓
FuelSiphonageDetector.process(event)
↓
InterestingEvent(category="fuel_drop")
↓
FuelAnomalyAgent.process(interesting_event)
↓
LLMClient.complete(system, user, response_model)
↓
Validated FuelAnomalyReport

```

## 4.1 Data Path Summary

1. Raw telematics rows are read from JSON/XLSX or an upstream fetch.
2. The adapter translates each row into a canonical `TelemetryReading`.
3. The detector keeps per-vehicle rolling state and fires when a drop pattern matches its rule.
4. The detector emits a compact `InterestingEvent` with evidence attached.
5. The agent retrieves a bounded set of historical cases.
6. The agent builds a prompt with the detector summary, evidence, and retrieved cases.
7. A pluggable LLM client is asked to fill a Pydantic schema.
8. The agent returns a frozen `FuelAnomalyReport`.

## 5. Canonical Schema Layer

The canonical schema is the core contract of the framework. It ensures everything after the adapter layer reasons over normalized types instead of provider-specific shapes.

### 5.1 BaseEvent

`BaseEvent` carries common metadata:

- `schema_version`
- `vehicle_id`
- `timestamp` in UTC
- `provider`
- `provider_event_id`
- `provider_extras` for traceability

### 5.2 TelemetryReading

`TelemetryReading` is the dominant canonical event type for polling-style providers such as Trakzee. It contains:

- position: latitude, longitude, altitude,
- motion: speed, heading, odometer,
- electrical state: external voltage, battery percentage,
- vehicle state: ignition and engine state,
- fuel fields: `fuel_level_percent`, `fuel_level_liters`, `fuel_level_raw`, `fuel_sensor_type`, `tank_capacity_liters`,
- environment: `ambient_temperature_c`,
- orientation: `pitch_deg`, `roll_deg`,
- human-readable context: `location_text`, `poi_text`.

The most important architectural improvement in the current codebase is that fuel sensing is now promoted into first-class canonical fields. Earlier designs left BLE fuel data buried in `provider_extras`, which made provider-agnostic detection impossible. The current schema resolves that.

### 5.3 DriverEvent

`DriverEvent` is the canonical discrete-event model for harsh braking, acceleration, turning, speeding, and ignition/idling transitions. It is structurally ready for future detectors even though the current feature path focuses on fuel anomalies.

### 5.4 InterestingEvent

`InterestingEvent` is the handoff contract between Tier 1 and Tier 2. Its design is intentionally small:

- `detector_name`
- `vehicle_id`
- `timestamp`
- `category` (currently `fuel_drop`)
- `severity`

- `summary`
- `confidence`
- `evidence` as detector-specific free-form detail
- `triggering_event_id`

This is a good separation point. The detector does not try to become an incident report; it only emits enough structured evidence for the agent to reason on.

## 5.5 CaseBase and IncidentCase

`IncidentCase` stores historical cases with symptoms, diagnosis, resolution, lessons, tags, region, and confidence. `CaseBase` is currently an in-memory store with simple category/region filtering and a placeholder embedding store. It is explicitly shaped so a future vector backend can preserve the same interface.

## 6. Adapter Layer

### 6.1 Adapter Contract

Every telematics provider should be normalized behind the abstract `TelematicsAdapter` contract. The contract is intentionally narrow: one `adapt(raw)` method that yields canonical events.

### 6.2 TrakzeeAdapter

`TrakzeeAdapter` is the reference implementation and currently the main production-grade adapter in the repo. It performs:

- IMEI-to-canonical vehicle mapping,
- local-time to UTC conversion,
- status normalization into `running` / `idle` / `stopped`,
- safe numeric coercion with null handling,
- meters-to-kilometers odometer conversion,
- BLE fuel JSON parsing from the Trakzee `Fuel` column,
- promotion of the first BLE fuel reading into `fuel_level_raw`,
- preservation of auxiliary channels and provider metadata in `provider_extras`.

Important nuance: Trakzee BLE fuel readings are treated as **raw counts**, not calibrated liters or percentages. The adapter intentionally refuses to fabricate calibrated values without per-vehicle calibration. That is the correct architectural decision.

The adapter is also structured to support future promotion of pitch/roll when those columns are enabled in the underlying Teltonika/Trakzee data profile. That is a good forward-compatible hook for reducing slope-related false positives.

### 6.3 Adapter Design Strengths

- Provider-specific quirks are isolated to one file.
- Missing values are normalized safely.
- Fuel data is promoted to canonical fields but raw details are still retained.
- The adapter avoids downstream assumptions about calibration.

### 6.4 Adapter Limitations

- Defaulting source timezone to Lusaka is convenient for the sample but dangerous if reused silently in other fleets.
- `provider_event_id` is synthesized as `imei:timestamp_seconds`, which can collide if multiple rows land in the same second.
- Multi-sensor trucks are flattened to the first BLE fuel channel for canonical use, which is pragmatic but lossy.

## 7. Tier 1 Detection Layer

### 7.1 Detector Abstractions

`Detector` is the base class for all Tier 1 rule-based logic. It depends on a `StateStore` protocol so detector state is decoupled from the storage backend. The default `InMemoryStateStore` is fine for tests and single-process demos, while a Redis-backed store can be added later without changing detector logic.

### 7.2 FuelSiphonageDetector Logic

The current detector implements a simple, explicit rule:

1. ignore non-telemetry events,
2. ignore events without `fuel_level_raw`,
3. cache per-vehicle baseline state,
4. only evaluate when ignition is off and engine state is `stopped` or `idle`,
5. compare current raw reading to the prior raw reading,
6. ignore increases or negative elapsed time,
7. ignore windows longer than `max_minutes`,
8. fire if relative drop exceeds `threshold_pct`.

The emitted evidence includes:

- drop percentage,
- prior and current raw values,
- elapsed minutes,
- engine and ignition state,
- fuel sensor type,
- optional altitude and orientation context.

### 7.3 Why the Detector Uses Raw Fuel

This is a strong design choice for real-world telematics. Many fleets do not have well-calibrated liters or percentages, especially when they retrofit BLE fuel probes. Using relative drop in raw counts allows the detector to function with lower-quality upstream data while still surfacing plausible siphonage candidates.

### 7.4 Detector Precision vs Recall Philosophy

The detector is intentionally recall-oriented. It does not try to fully rule out thermal contraction, slope settling, or other edge cases inside Tier 1. Instead, it sends compact evidence to Tier 2, where the agent can compare against historical cases. This separation of responsibilities is coherent.

### 7.5 Detector Strengths

- Small and understandable.
- State is explicit and backend-agnostic.
- Supports uncalibrated sensors.
- Adds altitude/pitch/roll context for future false-positive reasoning.

### 7.6 Detector Current Limits

- No route awareness or depot awareness.
- No fuel-station / refuel reconciliation.
- No per-vehicle baseline learning.
- No long-horizon aggregation for repeated small daytime siphonage.
- No direct handling of thermal contraction in Tier 1.
- No state expiration policy beyond time-window logic in the detector itself.

## 8. Tier 2 Agent Layer

---

### 8.1 LLMClient Protocol

The `LLMClient` protocol is the primary abstraction that keeps agent logic independent of model vendors. A client only needs one method:

```
complete(system: str, user: str, response_model: type[T]) -> T
```

That is a strong architectural boundary. Agent code does not need to know whether the provider is OpenAI, Anthropic, Gemini, Ollama, vLLM, or something else.

### 8.2 FuelAnomalyAgent

`FuelAnomalyAgent` is the concrete Tier 2 implementation for `fuel_drop` events. It works as follows:

1. decline any interesting event whose category is not `fuel_drop`,
2. retrieve up to `max_cases` from the case base,
3. build a user prompt that includes the detector summary, evidence, and formatted cases,
4. send system + user prompt to the configured `LLMClient`,
5. parse the LLM result into an internal verdict schema,
6. denormalize the verdict into a frozen `FuelAnomalyReport`.

### 8.3 Prompting Strategy

The system prompt is focused and conservative: it instructs the model to classify into one of `likely_siphonage`, `likely_false_positive`, or `uncertain`; cite case IDs; and prefer uncertainty over overconfident mistakes.

The user prompt contains:

- detector metadata,
- vehicle ID and timestamp,
- detector severity/confidence,
- free-form evidence key/values,
- the retrieved case descriptions including symptoms, diagnosis, resolution, and lessons.

### 8.4 FuelAnomalyReport

The report is intentionally denormalized and frozen. It stores:

- traceability fields to the detector event,
- assessment and confidence,
- summary and reasoning,
- recommended actions,
- referenced case IDs.

## 8.5 Agent Strengths

- Clear separation from provider SDKs.
- Schema-constrained output rather than free text.
- Reasoning is grounded on specific historical cases.
- Supports a mock client for zero-key demos and deterministic tests.

## 8.6 Agent Current Limits

- Case retrieval is currently simple category filtering, not real similarity search.
- The agent does not yet consume OEM manuals or structured route/depot context.
- No report persistence or workflow integration exists in the OSS repo yet.

# 9. LLM Provider Implementations

---

## 9.1 OpenALLMClient

The OpenAI client uses structured outputs via the OpenAI SDK parse path. The agent asks for a concrete Pydantic model and receives a validated instance back. That avoids JSON extraction hacks and is the cleanest provider implementation in the repo.

## 9.2 AnthropicLLMClient

The Anthropic client uses forced tool-use rather than a native structured-output parse endpoint. It generates the JSON schema from the requested Pydantic model, defines a tool with that schema, forces the model to call it, and validates the returned tool payload.

## 9.3 MockLLMClient

The mock client is not just a test convenience; it is also a product-enablement tool. It makes the quickstart and the Streamlit demo fully runnable without API keys and without external LLM spend.

# 10. Knowledge Layer

---

The seed knowledge pack is an important part of the current design. It includes both confirmed theft cases and false-positive cases, which is exactly what a good Tier 2 reasoning layer needs.

The cases cover patterns such as:

- large engine-off night-time siphonage,
- off-book refueling,
- thermal contraction false positives,
- mechanical siphonage via return-line tampering,
- small repeated daytime losses,
- slope-effect false positives referenced by the newer orientation-aware code paths.

This layer is currently synthetic and in-memory, but it already shapes the language and output style used by the agent.

# 11. Demo and UX Architecture

---

The repository already contains the kind of visual interface you described: a Streamlit app in `apps/demo/app.py` that lets a user upload a small Trakzee-shaped sample file or use bundled sample data.

## 11.1 What the Demo Does

- accepts `.json` and `.xlsx` inputs,
- enforces a small file-size cap,
- runs the adapter, detector, and agent pipeline,
- shows the canonical event sample,
- shows every Tier 1 firing and the detector evidence,
- shows the exact system prompt and user prompt sent to the LLM,
- shows final structured reports,
- supports OpenAI, Anthropic, and mock mode.

This is directly aligned with the intuition-building webpage concept: the demo makes the “black box” visible, especially the precise prompts and the case retrieval step.

## 11.2 Hosting Path

The demo is packaged for HuggingFace Spaces using Docker. The repo includes:

- a Dockerfile,
- a demo-specific requirements file,
- a deploy workflow that syncs `apps/demo` to a HuggingFace Space on pushes to `main`.

### 11.3 Current UX Gaps

- The demo is Streamlit-first rather than a custom product UI.
- It is intentionally scoped to small sample files and one main provider shape.
- It does not yet expose more advanced controls like threshold tuning, route context, or case-pack selection.

## 12. Testing Strategy

---

The current test suite is materially better than an early-stage scaffold. It covers:

- schema validation and immutability,
- Trakzee adapter transformation behavior,
- fuel-field promotion into canonical schema,
- interesting-event schema behavior,
- detector rule behavior and per-vehicle state isolation,
- orientation evidence propagation,
- agent prompt construction and report generation,
- mock LLM behavior,
- OpenAI and Anthropic client call construction via stubs,
- optional real-provider integration tests gated on API keys.

### 12.1 What Is Strong About the Tests

- They test the shape of the pipeline, not just isolated functions.
- They verify design-critical behavior, such as “no fabricated calibrated fuel values.”
- They include real integration hooks without forcing paid API usage in CI.

### 12.2 What Is Still Missing

- No persistence/storage backend tests because that backend is not implemented yet.
- No real route-planning, station-registry, or weather-join tests because those systems do not exist yet in OSS.
- No browser-level UI tests for the Streamlit demo.

## 13. Packaging, Tooling, and CI

---

### 13.1 Python and Packaging

The project targets Python 3.14 and is packaged with Hatchling. Optional extras exist for provider- or environment-specific capabilities such as `trakzee`, `llm`, `demo`, and `now docs`. Dev tooling is available both as a pip extra and as a uv dependency group.

### 13.2 CI Workflow

The CI workflow currently runs:

- Ruff linting,
- Ruff format checks,
- Mypy type checking,
- pytest on Python 3.14.

### 13.3 Demo Deployment Workflow

The repo also has a separate GitHub Actions workflow that auto-deploys the demo app to HuggingFace Spaces by syncing the `apps/demo` directory into the target Space repo.

## 14. Current Strengths

---

- The core architecture is now coherent end-to-end.
- Canonical fuel fields are properly modeled.
- The Tier 1/Tier 2 separation is technically defensible.
- LLM provider abstraction is clean and extendable.
- The demo app is unusually transparent and useful for visual intuition.
- The mock LLM path lowers friction for product demos and testing.
- The project is small enough to understand but already opinionated enough to be meaningful.

## 15. Current Weaknesses and Architectural Gaps

---

The repo is now functional, but it is still an alpha framework, not a complete fleet-intelligence platform.

- No storage backend for reports, state, or case embeddings beyond in-memory development implementations.
- No real similarity retrieval over the case base yet.
- No route/depot/station/weather integrations, which are important for high-precision fleet reasoning.
- No multi-detector or multi-agent orchestration yet.
- No CLI entrypoint despite the broader framework ambition.
- No authentication, tenancy, or production service API surface in the OSS repo.
- Current provider support is still narrow.
- Some logic is deliberately simple today and will need calibration per fleet.

## 16. Recommended Near-Term Evolution

---

1. Add vector-backed case retrieval while preserving the current `CaseBase` interface.
2. Add structured upstream context objects for route, depot, station registry, and weather joins.
3. Add a persistence layer for reports and detector state.
4. Expand provider adapters beyond Trakzee.
5. Generalize the Streamlit demo into a more product-like hosted sample-analysis page if desired.
6. Add a CLI and/or service API for non-demo operation.
7. Consider first-class support for multi-tank vehicles if the product focus deepens there.

## 17. Interpretation for Another Model or Engineer

---

If another model such as Claude is reviewing this codebase, the most important mental model is:

- **canonical schema first,**
- **cheap detector second,**
- **case-grounded LLM interpretation third,**
- **demo transparency as a product feature, not an afterthought**

The code should not be interpreted as a generic “LLM wrapper for fleet data.” It is a deliberately staged analytics system that tries to keep deterministic signal extraction and probabilistic interpretation separate. That separation is the central architectural idea of the repo.

## 18. Final Assessment

---

Siphyy is currently a credible alpha framework for fuel anomaly analysis in fleet telematics. Its strongest technical decisions are:

- canonical event normalization,
- stateful deterministic Tier 1 filtering,
- schema-constrained Tier 2 outputs,
- provider-agnostic LLM client abstraction,
- and a transparent demo experience that shows the exact reasoning inputs.

Its biggest remaining gaps are not conceptual confusion but missing production subsystems: durable storage, richer retrieval, more upstream business context, more adapters, and a more complete service surface.

Source basis for this report: repository code in `src/siphyy/`, `apps/demo/`, `examples/`, `tests/`, and GitHub workflow definitions present in the workspace on 2026-05-14.